От основ к мастерству

Предисловие Дэмиана Конвея

Изучаем глубже Perl



Рэндал Л. Шварц,
Брайан Д. Фой и Том Феникс

Intermediate Perl

Second Edition

Randal L. Schwartz, brian d foy
& Tom Phoenix

O'REILLY®

Perl
изучаем глубже

Второе издание

Рэндал Л. Шварц, Брайан Д. Фой
и Том Феникс



Санкт-Петербург — Москва
2008

Рэндал Л. Шварц, Брайан Д. Фой и Том Феникс

Perl: изучаем глубже, 2-е издание

Перевод А. Киселева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*О. Цилюрик*
Редактор	*В. Овчинников*
Корректура	*О. Макарова*
Верстка	*Д. Орлова*

Шварц Р., Фой Б., Феникс Т.
Perl: изучаем глубже, 2-е издание. – Пер. с англ. – СПб: Символ-Плюс, 2007. – 320 с., ил.
ISBN-13: 978-5-93286-093-9
ISBN-10: 5-93286-093-6

Книга «Perl: изучаем глубже» – продолжение мирового бестселлера «Learning Perl» («Изучаем Perl»), известного под названием «Лама». Издание поможет вам перешагнуть грань, отделяющую любителя от профессионала, и научит писать на Perl настоящие программы, а не разрозненные сценарии. Материал изложен компактно и в занимательной форме, главы завершаются упражнениями, призванными помочь закрепить полученные знания. Рассмотрены пакеты и пространства имен, ссылки и области видимости, создание и использование модулей. Вы научитесь с помощью ссылок управлять структурами данных произвольной сложности, узнаете, как обеспечить совместимость программного кода, написанного разными программистами. Уделено внимание и ООП, которое поможет повторно использовать части кода. Обсуждаются создание дистрибутивов, аспекты тестирования и передача собственных модулей в CPAN.

Книга адресована широкому кругу программистов, знакомых с основами Perl и стремящихся повысить свою квалификацию как в написании сценариев, так и в ООП, и призвана помочь им научиться писать эффективные, надежные и изящные программы.

ISBN-13: 978-5-93286-093-9
ISBN-10: 5-93286-093-6
ISBN 0-596-10206-2 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 19.10.2007. Формат 70×100 1/16. Печать офсетная.
Объем 20 печ. л. Тираж 2000 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Вступительное слово

Объектно-ориентированный механизм языка программирования Perl являет собой пример ловкости рук без всякого обмана. Он берет набор возможностей, не связанных с объектно-ориентированным стилем программирования, таких как пакеты, ссылки, хеши, массивы, подпрограммы и модули, и с помощью несложных заклинаний превращает их в полнофункциональные объекты, классы и методы.

Благодаря этому трюку программист, основываясь на своих познаниях языка Perl, может легко и просто перейти к объектно-ориентированному стилю программирования, не преодолевая горы нового синтаксиса и не переплывая океаны новых технологий. Это также означает, что объектно-ориентированные возможности языка Perl можно осваивать постепенно, по мере необходимости отбирая те, что наилучшим образом подходят для решения поставленных задач.

Однако здесь кроется одна проблема. Поскольку все эти пакеты, ссылки, хеши, массивы, подпрограммы и модули составляют основу объектно-ориентированного механизма, для использования объектно-ориентированных возможностей языка Perl необходимо знать принципы работы с пакетами, ссылками, хешами, подпрограммами и модулями.

Трудность именно в этом. Кривая обучения не исчезла, она всего лишь сократилась на полдесятка шагов.

Какие же *не*объектно-ориентированные возможности языка Perl надо изучить, чтобы можно было взяться за объектно-ориентированные?

Ответу на этот вопрос и посвящена данная книга. На ее страницах Рэндал, опираясь на 20-летний опыт работы с языком Perl и 40-летний опыт просмотра фильмов «Остров Джиллигана» и «Мистер Эд», описывает компоненты языка Perl, которые все вместе составляют фундамент его объектно-ориентированных возможностей. И, что еще лучше, на примерах показывает, как из этих компонентов создавать классы и объекты.

Итак, если объекты языка Perl вызывают у вас чувства, подобные тем, которые испытал Джиллиган на необитаемом острове, эта книга – как раз то, что доктор прописал.

Кроме того, вся информация в ней прямо из первых рук.

– Дэмиан Конвей (Damian Conway), май 2003

Предисловие

Более десяти лет тому назад (практически вечность по меркам Интернета) Рэндал Шварц написал первое издание книги «Learning Perl»[1]. За прошедшие годы сам Perl из «крутого» языка сценариев, используемого в первую очередь системными администраторами UNIX, вырос в полноценный объектно-ориентированный язык программирования, способный функционировать на практически любой платформе, известной человечеству.

Объем всех четырех изданий «Learning Perl» оставался практически неизменным (примерно 300 страниц), как в основном неизменным оставался и ее материал, рассчитанный на начинающих программистов. Однако времена изменились, и теперь о Perl можно рассказать значительно больше, чем когда появилось первое издание книги.

Рэндал назвал первое издание книги «Learning Perl Objects, References, and Modules», а сейчас книга получила название «Intermediate Perl» (Perl средней сложности), но на наш взгляд ей больше подошло бы название «Learning More Perl» (Изучаем Perl глубже)[2]. Данная книга продолжает обсуждение тем с того места, где оно было закончено в книге «Learning Perl». Здесь мы покажем вам, как писать большие программы на языке Perl.

Как и в «Learning Perl», мы старались сделать каждую главу настолько маленькой, чтобы ее можно было прочитать за час-другой. Каждая глава заканчивается серией упражнений, которые помогут вам на практических примерах закрепить только что прочитанный материал. Кроме того, в конце книги вы найдете приложение, в котором содержатся решения всех упражнений. Как и в «Learning Perl», материал этой книги подается в той же последовательности, что и в курсах обучения языку Perl, которые проводятся нами в компании Stonehenge Consulting Services.

---

1 Рэндал Шварц, Том Кристиансен «Изучаем Perl». – Пер. с англ. – BHV-Киев, 1999.

2 Не спрашивайте, почему книга не была названа именно так. Мы получили более 300 предложений по этой теме. На самом деле невозможно прекратить изучение Perl, поэтому название «Изучаем Perl глубже» фактически ничего не говорит о книге. Наш редактор выбрал название, которое говорит о том, чего следует ожидать от книги.

Чтобы извлечь максимум пользы из этой книги, вам необязательно быть гуру в UNIX и даже необязательно быть пользователем UNIX. Все, о чем говорится в этой книге, одинаково хорошо подходит как для Windows ActivePerl, так и для любой другой современной реализации. Чтобы иметь возможность пользоваться этой книгой, вам необходимо ознакомиться с книгой «Learning Perl» и гореть желанием продолжать двигаться вперед.

Структура книги

Данную книгу следует читать, начиная с первых глав, в том порядке, в каком они следуют, останавливаясь для выполнения упражнений. Материал каждой главы основан на предыдущих главах, и при обсуждении новой темы мы будем исходить из предположения, что предыдущие главы уже были вами прочитаны.

Глава 1 «Введение»

Содержит вводные положения.

Глава 2 «Основы»

Описывает некоторые промежуточные положения, знание которых потребуется при прочтении оставшейся части книги.

Глава 3 «Модули»

Описывает порядок работы с основными модулями Perl и с модулями сторонних производителей. Позже в этой же книге мы покажем, как создавать собственные модули, но в данной главе мы остановимся на использовании существующих модулей.

Глава 4 «Введение в ссылки»

Рассказывает о том, как организовать перенаправление, чтобы один и тот же программный код мог работать с различными наборами данных.

Глава 5 «Ссылки и области видимости»

Рассказывает о том, как Perl работает с указателями на данные, и дает краткое введение в анонимные структуры данных и автовивификацию.

Глава 6 «Управление сложными структурами данных»

Описывает создание структур данных с произвольной глубиной вложенности, включая массивы массивов и хеши хешей, обращение к ним и вывод их содержимого.

Глава 7 «Ссылки на подпрограммы»

Описывает поведение анонимных подпрограмм, которые могут создаваться динамически для последующего использования.

Глава 8 «Ссылки на дескрипторы файлов»

Описывает, как можно хранить дескрипторы файлов в скалярных переменных для передачи между различными частями программы или для сохранения в структурах данных.

Глава 9 «Практические приемы работы со ссылками»

Сложные операции сортировки, преобразование Шварца и работа с рекурсивно определенными данными.

Глава 10 «Разработка больших программ»

Рассматривает вопросы создания больших программ из нескольких файлов с программным кодом, разнесенным по разным пространствам имен.

Глава 11 «Введение в объекты»

Работа с классами, вызов методов, наследование и переопределение.

Глава 12 «Объекты и данные»

Экземпляры данных, конструкторы и методы доступа.

Глава 13 «Уничтожение объектов»

Описывает поведение объектов при уничтожении, включая объекты, существующие постоянно.

Глава 14 «Дополнительные сведения об объектах»

Множественное наследование, автоматические методы и ссылки на дескрипторы файлов.

Глава 15 «Экспортирование»

Как работает директива `use`, как определить, что нужно экспортировать, и как создать собственную процедуру импорта.

Глава 16 «Создание дистрибутивов»

Описывает порядок создания модулей, готовых к распространению, включая платформонезависимые инструкции по установке.

Глава 17 «Основы тестирования»

Описывает порядок тестирования программного кода с целью проверки его функциональности.

Глава 18 «Дополнительные сведения о тестировании»

Описываются более сложные аспекты тестирования программного кода и метаданных, такие как документация и покрытие тестами.

Глава 19 «Передача модулей в CPAN»

Описывает, как можно отправить свои разработки в CPAN.

Приложение A содержит решения всех упражнений.

Типографские соглашения

В книге приняты следующие соглашения по оформлению текста:

`Моноширинным шрифтом`

Выделены имена функций, модулей, файлов, переменных окружения, фрагменты программного кода и пр.

Курсивом

Выделены наиболее важные моменты и вновь вводимые термины.

Примеры программного кода

Данная книга призвана помочь вам в работе. Вы можете вставлять примеры программного кода из этой книги в свои приложения и в документацию, и для этого не надо обращаться в издательство O'Reilly за разрешением. Например, если вы пишете программу и заимствуете несколько отрывков программного кода из книги, вам не нужно обращаться за разрешением. Не требуется разрешение и для цитирования данной книги или примеров из нее при ответе на вопросы.

Если же вы собираетесь воспроизводить значительные фрагменты программного кода из этой книги (например, в документации), то разрешение необходимо. Разрешение нужно получить и в случае, если вы планируете продавать или распространять компакт-диски с примерами из этой книги.

Мы не требуем добавлять ссылку на первоисточник при цитировании (но совсем не против этого). Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN, например «Intermediate Perl, by Randal L. Schwartz, Brian D. Foy, and Tom Phoenix. Copyright 2006 O'Reilly Media, Inc., 0-596-10206-2».

За получением разрешения на использование значительных объемов программного кода примеров из этой книги обращайтесь по адресу *permissions@oreilly.com*.

Отзывы и предложения

С вопросами и предложениями, касающимися этой книги, обращайтесь в издательство:

O'Reilly Media

1005 Gravenstein Highway North

Sebastopol, CA 95472

(800) 998-9938 (в Соединенных Штатах Америки или в Канаде)

(707) 829-0515 (международный)

(707) 829-0104 (факс)

Список опечаток, файлы с примерами и другую дополнительную информацию вы найдете на сайте книги:

http://www.oreilly.com/catalog/intermediateperl

Свои пожелания и вопросы технического характера отправляйте по адресу:

bookquestions@oreilly.com

Дополнительную информацию о книгах, обсуждения, центр ресурсов издательства O'Reilly вы найдете на сайте:

http://www.oreilly.com

Safari Enabled



Если на обложке книги есть пиктограмма «Safari® Enabled», это означает, что книга доступна в Сети через O'Reilly Network Safari Bookshelf.

Safari предлагает намного лучшее решение, чем электронные книги. Это виртуальная библиотека, позволяющая без труда находить тысячи лучших технических книг, вырезать и вставлять примеры кода, загружать главы и находить быстрые ответы, когда требуется наиболее верная и свежая информация. Она свободно доступна по адресу *http://safari.oreilly.com*.

Благодарности

Рэндал. В предисловии к первому изданию книги «Learning Perl» я выразил свою признательность Бивертону Мак-Менамину (Beaverton McMenamin) – владельцу паба Cedar Hills (Кедровые холмы), что находится рядом с моим домом, за бесплатно предоставленный кабинет, где я имел обыкновение писать черновики книги на моем Powerbook 140. Этот паб стал для меня талисманом, приносящим удачу. Практически все свои книги (включая и эти слова) я писал здесь и очень надеюсь, что удача мне не изменит и на этот раз!

Теперь в этом пабе подают прекрасное пиво, сваренное тут же в маленькой пивоварне, и сладкие сэндвичи, но пропала моя любимая хлебная пицца, которую заменили ежевичным коктейлем (местный рецепт) и острой джамбалайей. (Кроме того, здесь появились два новых кабинета и несколько столов.) Ну а вместо Powerbook 140 у меня теперь более современный Titanium Powerbook, у которого диск в 1000 раз больше, оперативной памяти в 500 раз больше и процессор в 200 раз быстрее. На нем установлена полноценная UNIX-подобная операционная система (OS X) вместо ограниченной версии Mac OS. Все свои черновики (включая и этот) я отправляю через модем сотового телефона на скорости 144 K, могу напрямую общаться со своими рецензентами, и мне не нужно возвращаться домой к моему модему, передающему данные по телефонной линии со скоростью 9600 бод. Как изменились времена!

Еще раз большое спасибо всем, кто работает в «Кедровых холмах», за их неизменное гостеприимство.

Как и в четвертом издании книги «Learning Perl», я должен отметить, что многим обязан своим студентам из Stonehenge Consulting Services за их каверзные (?) вопросы, которые появлялись, когда сложность материала превышала уровень их подготовки. Благодаря им я смог продолжить совершенствование материала, который лег в основу этой книги.

Следует заметить, что все началось с полудневного курса «Что нового в Perl 5?» Марджи Левин (Margie Levine) из Silicon Graphics, а также моего собственного четырехдневного курса «Лама» (Llama) (в то время основанного на Perl версии 4). Со временем у меня возникла идея превратить эти короткие заметки в полноценный курс и подтолкнуть сотрудника компании Stonehenge Джозефа Холла (Joseph Hall) к участию в решении этой задачи. (Он один из тех, кто отбирал примеры программного кода для курса.) Джозеф разработал двухдневный курс для Stonehenge и одновременно написал прекрасную книгу «Effective Perl Programming», которая затем стала использоваться как учебник.

За эти годы в разработке курса «Пакеты, ссылки, объекты и модули» принимали участие многие преподаватели из Stonehenge, включая Чипа Зальценберга (Chip Salzenberg) и Тэда Мак-Клеллана (Ted McClellan). Но большая часть изменений и дополнений была внесена Томом Фениксом (Tom Phoenix), который становился «служащим месяца» в компании Stonehenge настолько часто, что мне, наверное, придется уступить ему мое привилегированное место на парковке. Том отлично управляется с материалами (как Тэд управляется с делами), благодаря чему я могу спокойно сосредоточиться на своих обязанностях президента и дворника компании Stonehenge.

Том Феникс написал большую часть примеров для этой книги и своевременно предоставлял свои рецензии во время моей работы над книгой. Его рукой были написаны целые абзацы, так что мне оставалось только вставлять их на место той бессмыслицы, что была написана мною. У нас получилась отличная команда, которая дружно работает и в аудитории, и над книгой. Именно за приложенные усилия мы признали Тома соавтором, но я готов взять всю вину на себя, если какая-либо часть книги вам не понравится, потому что, скорее всего, в этом нет вины Тома.

И последний, но не в последнюю очередь, кому я хотел бы выразить свою благодарность, – это Брайан Д. Фой (brian d foy), который примкнул к работе над книгой, начиная со второго ее издания, и предложил массу изменений и дополнений к этому изданию.

Разумеется, книга не состоялась бы, не будь темы для обсуждения и каналов распространения, поэтому я хочу выразить свою признательность Ларри Уоллу (Larry Wall) и Тиму O’Рейли (Tim O’Reilly).

Спасибо вам, ребята, за то что вы создали компанию, которая оплачивала мне мои затраты в течение 15 последних лет.

И, как обычно, отдельное спасибо Лейле и Джеку, которые научили меня всему, что я знаю о писательской деятельности, и убедили меня в том, что я должен быть чуть большим (?), чем программист, который умеет писать. Благодаря им я стал писателем, который умеет программировать. Спасибо вам.

Спасибо и вам, уважаемый читатель. Именно для вас я трудился долгие часы, потягивая холодное пиво и поедая пудинг, стараясь не залить клавиатуру моего ноутбука. Спасибо вам за то, что вы читаете мою книгу. Я искренне надеюсь, что внес свой вклад (пусть и незначительный) в ваше образование. Если вы встретите меня когда-нибудь на улице, просто скажите: «Привет!».[1] Мне это будет приятно. Спасибо вам.

Брайан. В первую очередь я хотел бы сказать спасибо Рэндалу, так как впервые я познакомился с Perl благодаря первому изданию его книги «Learning Perl» и многое узнал, преподавая курсы «Лама» и «Альпака» в компании Stonehenge Consulting. Учить других – часто лучший способ научиться самому.

Мне удалось убедить Рэндала в необходимости обновить книгу «Learning Perl», а когда эта работа была закончена, я заметил ему, что пора обновить и эту книгу. Наш редактор Элисон Рэндал (Allison Randal) согласилась с этим и приложила максимум усилий, чтобы не нарушить наш график.

Отдельное спасибо Стейси (Stacey), Бастеру (Buster), Мими (Mimi), Роско (Rosco), Амелии (Amelia), Лиле (Lila) и всем тем, кто пытался отвлечь меня от работы, когда я был занят.

От нас обоих. Спасибо нашим рецензентам: Дэвиду Адлеру (David H. Adler), Стефену Дженкинсу (Stephen Jenkins), Кевину Мельтцеру (Kevin Meltzer), Мэттью Масгрову (Matthew Musgrove), Эндрю Сэвиджу (Andrew Savige) и Риккардо Сигнесу (Ricardo Signes) за их комментарии к рукописи этой книги.

Спасибо нашим студентам, которые помогли нам понять, какие части курса необходимо пересмотреть и дополнить. Именно благодаря вам мы испытываем чувство гордости за свою работу.

Спасибо членам группы Perl Mongers, кто принимал нас в своих городах как родных. Давайте встретимся еще когда-нибудь.

И наконец, огромное спасибо Ларри Уоллу за его большие и мощные игрушки, которые позволили нам сделать свою работу намного быстрее, проще и с увлечением.

---

[1] К тому же вы можете спросить меня что-нибудь о Perl. Я не возражаю.

1

Введение

Добро пожаловать в следующий этап изучения языка программирования Perl. Вероятно, вы здесь или для того, чтобы научиться писать программы длиннее 100 строк, или по принуждению своего босса.

Наша предыдущая книга «Learning Perl» была такой большой, потому что она представляет собой введение в язык программирования Perl и его использование для написания маленьких и средних программ (которые, по нашим наблюдениям, составляют большую часть кода, написанного на языке Perl). Чтобы избежать увеличения объема книги под кодовым названием «Лама», мы намеренно и очень аккуратно оставили за бортом довольно много информации.

На следующих страницах вы найдете «продолжение истории», изложенной в том же дружественном стиле. Она содержит сведения, знать которые совершенно необходимо, если вы собираетесь писать программы длиной от 100 до 10 000 строк.

Например, вы узнаете, как организовать коллективную работу над проектом. Это просто здорово, потому что если вы не собираетесь работать по 35 часов каждый день, вам наверняка потребуется помощь при решении крупных задач. Вам также придется обеспечивать совместимость программного кода, написанного различными программистами, чтобы свести результаты коллективного труда в единое приложение.

В этой книге также показано, как работать с большими и сложными структурами данных, которые мы небрежно называем «хешами хешей», или «массивами массивов хешей массивов». Познакомившись со ссылками поближе, вы сможете управлять структурами данных произвольной сложности.

Затем мы перейдем к заслуживающим внимания понятиям объектно-ориентированного программирования (ООП), которое поможет вам повторно использовать части своего (а может быть, и чужого) программно-

го кода с минимальными переделками. Вы без труда разберетесь в этой теме, даже если никогда раньше не сталкивались с объектами.

Немаловажным аспектом коллективной разработки является цикличность выпуска новых версий и разработка тестов для проведения модульного и интеграционного тестирования. Здесь вы получите основные сведения о создании дистрибутивных пакетов и разработке модульных тестов, которые могут применяться как в процессе работы над приложением, так и для окончательного тестирования готового продукта.

И наконец, точно так же, как и в предыдущих изданиях «Learning Perl», мы будем развлекать вас интересными примерами и плохими каламбурами. (Мы отправили-таки Фреда (Fred), Барни (Barney), Бетти (Betty) и Уилму (Wilma) по домам. А на новые роли пригласили звезд.)

Что вы должны знать?

Предполагается, что вы уже прочитали книгу «Learning Perl» или по крайней мере делаете вид, что уже достаточно наигрались с Perl и обладаете базовыми знаниями. В этой книге, например, не рассказывается, как обращаться к элементам массива или как вернуть некоторое значение из подпрограммы.

Как минимум вы должны знать:

- Как запускать в своей системе программы, написанные на Perl
- Три основных типа переменных в Perl: скаляры, массивы и хеши
- Конструкции управления ходом исполнения, такие как `while`, `for` и `foreach`
- Что такое подпрограммы
- Операторы языка Perl, такие как `grep`, `map`, `sort` и `print`
- Функции работы с файлами, такие как `open`, функции чтения из файлов и `-X` (тестирование файлов)

В этой книге вы сможете получить более глубокие знания по данным темам, но мы полагаем, что основы вы уже знаете.

Как быть со сносками?

Как и в книге «Learning Perl», некоторые дополнительные сведения, знание которых не обязательно при первом прочтении, оформлены в виде сносок.[1] При первом прочтении их можно пропустить, но при повторном было бы желательно прочитать и их. В сносках нет ничего такого, что было бы необходимо для понимания остального материала.

[1] Таких, как эта.

Как быть с упражнениями?

Тренировки помогают усвоить материал. А лучший способ потренироваться – выполнить несколько упражнений после каждого получасового изучения очередной темы. Разумеется, если вы читаете достаточно быстро, то до конца главы доберетесь немного быстрее, чем за полчаса. Приостановитесь, сделайте передышку и выполните упражнения!

Каждое упражнение едва ли займет больше пары минут. Это время соответствует середине колоколообразной кривой, однако будет совсем неплохо, если у вас это займет чуть больше или чуть меньше времени. Иногда время решения упражнений зависит лишь от того, насколько часто вам приходилось сталкиваться с решением подобных задач. Так что это время – всего лишь ориентир.

Ответы к упражнениям приведены в приложении. Однако попробуйте не заглядывать туда, иначе вы снизите обучающую ценность упражнения.

Что делать, если я преподаю Perl?

Если вы преподаете язык программирования Perl и решили использовать эту книгу в качестве учебного пособия, то должны понимать, что каждый набор упражнений слишком короток для большинства студентов, чтобы занять их на полные 45 минут. Одни упражнения отнимут больше времени, другие меньше. Мы обнаружили этот факт лишь после того, как расставили эти маленькие числа в квадратных скобках.

Итак, приступим. Обучение начнется, как только вы перевернете страницу…

2

Основы

Прежде чем приступить к изучению основного материала, рассмотрим некоторые понятия языка Perl, которыми мы будем оперировать в этой книге и которые обычно не попадают в фокус внимания начинающих программистов. Попутно мы представим персонажей, которые встретятся нам в этой книге.

Операторы списков

Вы уже наверняка знаете несколько операторов Perl, предназначенных для работы со списками, но скорее всего вы и предположить не могли, что эти операторы работают именно со списками. Из них чаще остальных, пожалуй, применяется оператор `print`. Он может принимать один или несколько аргументов и связывает их в единое целое.[1]

```
print 'Кораблекрушение потерпели двое ', 'Джиллиган', ' и ', 'Шкипер', "\n";
```

Для работы со списками предназначен еще целый ряд операторов, о которых вы уже узнали из книги «Learning Perl». Оператор `sort` упорядочивает входной список. В известной песне[2] потерпевшие кораблекрушение упоминаются не в алфавитном порядке, однако мы можем исправить этот недостаток с помощью оператора `sort`.

```
my @castaways = sort qw(Джиллиган Шкипер Джинджер Профессор Мери-Энн);
```

---

[1] В этой книге текстовые константы в программном коде переведены на русский язык. Работоспособность кода была проверена в трех операционных системах – Windows 98, Linux Mandriva 2006 и QNX 6.3.0 – в тех реализациях Perl 5.xx, которые были последними для каждой из ОС на момент перевода книги. – *Примеч. науч. ред.*

[2] В балладе об острове Джиллигана («The Ballad of Gilligan's Isle»), написанной Джорджем Уайлом (George Wyle) и Шервудом Шварцем (Sherwood Schwartz).

Оператор reverse возвращает список в обратном порядке.

```
my @castaways = reverse qw(Джиллиган Шкипер Джинджер Профессор Мери-Энн);
```

В языке Perl имеется масса других операторов, которые работают со списками, и, как только вы научитесь использовать их, вы обнаружите, что объем ввода с клавиатуры уменьшился, а ваши намерения стали выражаться более ясно.

Фильтрация списков с помощью grep

Оператор grep принимает список значений и «условное выражение». Он извлекает из списка одно значение за другим и помещает их в переменную $_. После этого производится вычисление условного выражения в скалярном контексте. Если в результате получается «истина», grep передает значение $_ в выходной список.

```
my @lunch_choices = grep &is_edible($_), @gilligans_posessions.
```

В списочном контексте оператор grep возвращает список всех прошедших проверку элементов, а в скалярном – количество отобранных элементов.

```
my @results = grep EXPR, @input_list;
my $count = grep EXPR, @input_list;
```

В данном случае EXPR означает любое скалярное выражение, которое должно выполнить проверку значения переменной $_ (явно или неявно). Например, чтобы отыскать все числа больше 10, в условном выражении можно сравнить переменную $_ со значением 10.

```
my @input_numbers = (1, 2, 4, 8, 16, 32, 64);
my @bigger_than_10 = grep $_ > 10, @input_numbers;
```

В результате будет получена последовательность чисел 16, 32 и 64. Здесь имеет место явное обращение к переменной $_. В следующем примере показано косвенное обращение к этой переменной из оператора поиска по шаблону:

```
my @end_in_4 = grep /4$/, @input_numbers;
```

Теперь мы получим числа 4 и 64.

Оператор grep запоминает оригинальное значение переменной $_ и восстанавливает его по окончании работы. Переменная $_ – это не просто копия элемента данных, на самом деле ее можно рассматривать как псевдоним фактического элемента, чем-то похожего на переменную цикла foreach.

Если условное выражение достаточно сложное, его можно оформить в виде подпрограммы:

```
my @odd_digit_sum = grep digit_sum_is_odd($_), @input_numbers;

sub digit_sum_is_odd {
    my $input = shift;
```

```
    my @digits = split //, $input; # Предполагается, что элемент списка
                                   # содержит только цифровые символы
    my $sum;
    $sum += $_ for @digits;
    return $sum % 2;
}
```

Теперь мы получим список, содержащий числа 1, 16 и 32. Суммы цифр этих чисел при делении на 2, выполняемом в последней строке подпрограммы, дают остаток 1, что расценивается оператором grep как «истина».

Синтаксис оператора grep имеет две формы. Только что мы продемонстрировали форму выражения, а сейчас покажем блочную. Применяя такую форму записи вместо явного определения подпрограммы, которая служит для проверки в единственном месте, мы можем поместить тело подпрограммы прямо в оператор grep:[1]

```
my @results = grep {
    блок;
    программного;
    кода;
} @input_list;

my $count = grep {
    блок;
    программного;
    кода;
} @input_list;
```

Точно так же, как и при записи в форме выражения, оператор grep на время помещает очередной элемент входного списка в переменную $_. Затем исполняется блок программного кода. Результат последнего выражения в блоке определяет конечный результат выполнения всего блока. (Как и любое другое условное выражение, оно вычисляется в скалярном контексте.) Поскольку теперь у нас имеется целый блок, мы можем внутри него работать с переменными, область видимости которых будет ограничена данным блоком. Перепишем предыдущий пример, придерживаясь блочного синтаксиса:

```
my @odd_digit_sum = grep {
    my $input = $_;
    my @digits = split //, $input; # Предполагается, что элемент списка
                                   # содержит только цифровые символы
    my $sum;
    $sum += $_ for @digits;
    $sum % 2;
  } @input_numbers;
```

[1] В блочной форме записи символ запятой между блоком и входным списком не ставится. Когда оператор grep записывается в форме выражения, запятая между условным выражением и списком ставится обязательно.

Обратите внимание на два отличия: входные значения теперь извлекаются из переменной `$_`, а не из списка переданных аргументов, а кроме того, мы убрали оператор `return`. Оставить его было бы ошибкой, потому что теперь мы находимся уже не внутри подпрограммы, а в блоке кода.[1] Разумеется, мы можем оптимизировать этот блок, отказавшись от промежуточных переменных:

```
my @odd_digit_sum = grep {
    my $sum;
    $sum += $_ for split //;
    $sum % 2;
  } @input_numbers;
```

Не бойтесь описывать ход исполнения более явно, если это поможет вам и вашим коллегам точнее понимать и сопровождать программный код. Это главный приоритет.

Преобразование списка с помощью оператора map

Оператор `map` обладает похожим синтаксисом, и во многом его действия напоминают оператор `grep`. Например, он временно помещает элементы списка друг за другом в переменную `$_` и допускает две формы записи: в форме выражения и в форме блока.

Однако это *выражение отображения* (*mapping expression*), а не условное выражение. Выражение оператора `map` вычисляется в списочном контексте (а не в скалярном, как в `grep`). Каждое обращение к выражению дает часть полного результата. Полный результат представляет собой список из частных результатов. В скалярном контексте `map` возвращает количество элементов, возвращаемых в списочном контексте. Однако оператор `map` очень редко встречается в скалярном контексте (если вообще встречается).

Начнем с простого примера:

```
my @input_numbers = (1, 2, 4, 8, 16, 32, 64);
my @result = map $_ + 100, @input_numbers;
```

Оператор `map` поместит каждый из семи элементов списка в переменную `$_`, в результате мы получим список, где каждому входному числу будет соответствовать единственный результат – число, которое больше исходного на 100. Таким образом, в списке `@result` будут находиться числа 101, 102, 104, 108, 116, 132 и 164.

Однако каждому элементу входного списка может соответствовать и несколько элементов в выходном списке. Посмотрим, что произойдет, если в результате обработки каждого входного элемента будут получаться два выходных элемента:

---

[1] Вызов оператора `return` привел бы к выходу из подпрограммы, которая содержит данный блок кода. Некоторые из нас сталкивались с этой ошибкой в своих первых работах.

```
my @result = map { $_, 3 * $_ } @input_numbers;
```

Теперь каждому входному элементу у нас соответствуют два выходных: 1, 3, 2, 6, 4, 12, 8, 24, 16, 48, 32, 96, 64 и 192. Мы можем сохранить эти пары чисел в виде хеша, если потребуется определить значения чисел в три раза больших степеней двойки:

```
my %hash = @result;
```

Или, если обойтись без промежуточных переменных:

```
my %hash = map { $_, 3 * $_ } @input_numbers;
```

Как видите, оператор map отличается высокой степенью гибкости – он позволяет создать для каждого входного элемента произвольное число выходных элементов. При этом для каждого входного элемента число выходных элементов может быть и разным. Посмотрим, что произойдет, если мы попытаемся разбить числа на отдельные цифры:

```
my @result = map { split //, $_ } @input_numbers;
```

Внутри блока числа разделяются на цифры. Для чисел 1, 2, 4 и 8 мы получим по одному результату. Для чисел 16, 32 и 64 – по два. Когда оператор map объединит результаты, получится список 1, 2, 4, 8, 1, 6, 3, 2, 6 и 4.

Если в результате вычисления выражения для некоторого элемента входного списка получается пустой список, оператор map просто не вставляет его в конечный результат. Эту особенность можно использовать для выборки определенных элементов. Допустим, что нам надо отобрать только отделенные (см. выше) цифры, если последняя из них 4:

```
my @result = map {
    my @digits = split //, $_;
    if ($digits[-1] == 4) {
        @digits;
    } else {
        ( );
    }
} @input_numbers;
```

Если она равна 4, то в результате оценки выражения @digits (в списочном контексте) возвращаются отделенные цифры. Если же не равна, возвращается пустой список, и результат для заданного входного элемента удаляется. Таким образом, оператор map может применяться вместо оператора grep, но не наоборот.

Разумеется, все, что можно сделать с помощью операторов grep и map, можно сделать и с помощью циклов foreach. Но точно так же можно программировать и на ассемблере или даже в машинных кодах.[1] Дело не в том, как можно реализовать тот или иной алгоритм, а в том, что

[1] Если вы достаточно давно работаете с вычислительной техникой, чтобы знать о существовании машинных кодов.

операторы grep и map позволяют уменьшить сложность программы и сконцентрироваться на решении задач высокого уровня, не отвлекаясь на детали.

Организация ловушек ошибок с помощью eval

Нередко исполнение обычных строк программного кода приводит к аварийному завершению программы, если что-то идет не так, как ожидалось.

```
my $average = $total / $count;  # деление на ноль?
print "okay\n" unless /$match/; # ошибочный шаблон?

open MINNOW, '>ship.txt'
or die "Невозможно создать файл 'ship.txt': $!";  # недостаточно прав?

&implement($_) foreach @rescue_scheme; # ошибка внутри подпрограммы?
```

Однако незапланированный ход событий вовсе не означает, что надо смириться с аварийным завершением программы. Для вылавливания разного рода ошибок Perl предоставляет оператор eval.

```
eval { $average = $total / $count } ;
```

Если во время исполнения блока eval произойдет ошибка, это уже не приведет к завершению всей программы, просто управление будет передано строке, следующей сразу же за блоком eval. Обычно после исполнения оператора eval проверяется значение переменной $@, которая будет содержать пустое значение (в случае отсутствия ошибки) или строку с сообщением об ошибке, например «divide by zero».

```
eval { $average = $total / $count } ;
print "Продолжение после ошибки: $@" if $@;

eval { &rescue_scheme_42 } ;
print "Продолжение после ошибки: $@" if $@;
```

Сразу за блоком eval необходимо ставить символ точка с запятой, поскольку eval – это функция (а не управляющая конструкция, как, например, if или while). Но сам блок является самым настоящим блоком, и в нем можно определять локальные переменные (переменные my) и любые другие операторы. Как и любая другая функция, eval имеет возвращаемое значение (значение последнего выражения или значение, возвращаемое оператором return). Разумеется, если в процессе исполнения блока кода произойдет ошибка, никакого значения возвращено не будет, то есть будет получено значение undef в скалярном контексте или пустой список в списочном. Это позволяет оформлять безопасное вычисление выражений так:

```
my $average = eval { $total / $count } ;
```

Теперь переменная $average будет содержать либо частное от деления, либо значение undef в зависимости от того, насколько успешно завершилась операция деления.

Perl поддерживает даже вложенные блоки eval. Это позволяет отлавливать ошибки во вложенных подпрограммах. Но eval не может перехватывать фатальные ошибки, которые приводят к краху самого Perl. Сюда можно отнести получение сигнала,[1] который нельзя перехватить, ошибка нехватки памяти и прочие катастрофические ситуации. Оператор eval не может применяться и для вылавливания синтаксических ошибок, поскольку компиляция блока eval производится одновременно с остальной частью программы, а не во время исполнения. Он не может перехватывать предупреждения (хотя Perl дает такую возможность с помощью $SIG{__WARN__}).

Исполнение программного кода, созданного динамически

Существует еще одна форма обращения к оператору eval, когда в качестве параметра выступает не блок кода, а строка. В этом случае компиляция и исполнение строки производятся уже во время исполнения программы. Такая возможность выглядит очень удобной и полезной, но она чрезвычайно опасна, если в строку могут попасть данные, полученные из ненадежного источника. Мы не рекомендуем вычислять строковые выражения с помощью оператора eval, разве что в исключительных случаях. Мы рассмотрим эту возможность чуть позже, и, кроме того, вы наверняка встретите такую форму записи оператора eval в чужих программах, поэтому мы покажем, как она применяется на практике:

```
eval '$sum = 2 + 2';
print "Сумма = $sum\n";
```

Perl выполнит это выражение в контексте окружающего программного кода, т. е. практически точно так же, как если бы это выражение было оформлено обычным образом. Результатом оператора eval является результат вычисления последнего выражения, поэтому совершенно необязательно вставлять весь блок операторов в строку eval.

```
#!/usr/bin/perl

foreach my $operator ( qw(+ - * /) ) {
    my $result = eval "2 $operator 2";
    print "2 $operator 2 = $result\n";
}
```

Здесь в цикле выполняется перебор операторов + - * /, и каждый из них поочередно работает внутри блока кода оператора eval. В строке, которая передается оператору eval, значение переменной $operator интерполируется в строку. После этого eval исполняет программный код, заключенный в строковую переменную, и возвращает результат последнего выражения, который записывается в переменную $result.

---

[1] Имеются в виду сигналы UNIX, например SIGKILL. – *Примеч. науч. ред.*

Если eval не сможет скомпилировать и исполнить программный код строки, которая ему была передана, мы получим точно такое же значение переменной $@, как и в случае использования eval в блочной форме. В следующем примере мы хотели перехватить ошибку деления на ноль, но в результате не смогли разделить число на «ничто» (еще одна разновидность ошибки).

```
print 'Частное: ', eval '5 /', "\n";
warn $@ if $@;
```

Оператор eval перехватит синтаксическую ошибку и запишет сообщение в переменную $@, что мы проверяем сразу же после вызова eval.

```
Частное:
syntax error at (eval 1) line 2, at EOF
```

Далее, в главах 10, 17 и 18, мы рассмотрим применение eval для загрузки необязательных модулей. Если Perl оказывается не в состоянии загрузить какой-либо модуль, он обычно останавливает исполнение программы. Мы будем перехватывать эту ошибку и продолжать работу программы.

На всякий случай, если вы пропустили наше предыдущее предупреждение, напомним еще раз: используйте эту форму оператора eval только в исключительных случаях. Если есть возможность выполнить те же самые действия другим способом, попробуйте сначала реализовать его. Мы обратимся к данной возможности в главе 10 для загрузки программного кода из внешнего файла, но при этом мы продемонстрируем другой, более интересный способ сделать то же самое.

Упражнения

Ответы на эти вопросы вы найдете в приложении A в разделе «Ответы к главе 2».

Упражнение 1 [15 мин]

Напишите программу, которая принимала бы из командной строки список файлов и отбирала бы с помощью оператора grep те из них, размер которых не превышает 1000 байт. Воспользуйтесь оператором map, чтобы перед каждым полученным в результате именем файла вставить четыре пробела и символ перевода строки после имени. Выведите получившийся список.

Упражнение 2 [25 мин]

Напишите программу, которая просила бы пользователя ввести шаблон (регулярное выражение). Строка должна вводиться с клавиатуры, а не приниматься в виде аргумента командной строки. Из заранее определенного каталога (например, */etc* или *c:\\windows*) выведите спи-

сок файлов, чьи имена совпадают с введенным шаблоном. Программа должна продолжать работу до тех пор, пока пользователь не введет в качестве шаблона пустую строку. Пользователь не должен вводить символы слэша, которые в Perl традиционно служат для разграничения шаблонов. Вводимые шаблоны должны разделяться символом перевода строки. Постарайтесь обеспечить устойчивость программы к ошибкам в шаблонах, таким как отсутствие парных скобок.

3

Модули

Модули – это строительные блоки наших программ. Они содержат вспомогательные подпрограммы, переменные и даже классы. По ходу обучения строительству собственных модулей рассмотрим некоторые стандартные модули, которые могут представлять для вас интерес. Мы также рассмотрим основы работы с модулями, которые написали другие разработчики.

Стандартный дистрибутив

Самые популярные модули уже входят в дистрибутивы Perl. В самых современных дистрибутивах модули занимают более 50 мегабайт. В дистрибутиве Perl 5.003_07, выпущенном в октябре 1996 года, количество модулей составило 98. На начало 2006 года в дистрибутиве Perl 5.8.8 их было уже 359.[1] Одно из преимуществ Perl состоит в том, что он поставляется с большим количеством готового к использованию программного кода, избавляя вас от необходимости самостоятельно выполнять значительные объемы работ.

Мы будем стараться указывать, какие модули включаются в дистрибутив Perl (и о большинстве из них попутно будем уточнять, начиная с какой версии эти модули вошли в дистрибутив). Мы будем называть эти модули «базовыми» или указывать, что они входят в «стандартный дистрибутив». Если Perl установлен, значит, должны иметься и эти модули. В процессе создания книги мы работали с Perl 5.8.7, поэтому будем считать эту версию текущей.

---

[1] Прочитав книгу, вы научитесь пользоваться модулем `Module::CoreList`, с помощью которого сможете проверить количество модулей в своем дистрибутиве. Именно с его помощью мы получили числа, которые приведены в тексте.

Не исключено, что в своем коде вы задействуете только базовые модули, чтобы гарантировать его работоспособность в любом окружении, где, по крайней мере, установлен Perl той же версии, что и у вас.[1] В данной книге не будет дискуссий на эту тему, поскольку мы любим репозитарий модулей CPAN слишком сильно, чтобы обойтись без него.

Использование модулей

Практически все модули Perl поставляются вместе с документацией, и даже если внутреннее устройство модуля нам неизвестно, это не должно нас волновать, если только мы знаем, как пользоваться его интерфейсом. В конце концов, интерфейсы для того и существуют, чтобы скрывать детали реализации.

Чтобы почитать документацию к модулю на своей машине, можно воспользоваться командой `perldoc`, передав ей имя модуля.

```
$ perldoc File::Basename

NAME
          fileparse - split a pathname into pieces

          basename - extract just the filename from a path

          dirname - extract just the directory from a path

SYNOPSIS

          use File::Basename;

          ($name,$path,$suffix) = fileparse($fullname,@suffixlist)
          fileparse_set_fstype($os_string);
          $basename = basename($fullname,@suffixlist);
          $dirname = dirname($fullname);
```

Мы привели лишь начало описания, чтобы показать вам самый важный раздел (по крайней мере, самое главное, что поможет начать работу с модулем). Обычно документация к модулям оформляется в формате страниц справочного руководства ОС UNIX и начинается с разделов NAME (название) и SYNOPSIS (краткое описание).

В кратком описании приводятся полезные примеры работы с модулем. То есть некоторые методики и синтаксис Perl могут быть незнакомы вам, но, руководствуясь примерами, вы сможете заставить свою программу работать.

Модули могут обладать интерфейсами различных типов, учитывая, что Perl представляет собой смесь процедурного, функционального и объектно-ориентированного стилей программирования. Мы будем

---

[1] Возможно, это преждевременно, но мы заметим, что наряду с другими историческими сведениями в модуле `Module::CoreList` имеются списки модулей для каждой из предыдущих версий Perl.

использовать эти модули несколько различными способами, но пока у нас имеется описание модуля, мы не должны испытывать каких-либо проблем.

Функциональные интерфейсы

Для загрузки модулей[1] предназначена встроенная директива use. Мы пока не будем обсуждать все тонкости ее применения, подробно об этом мы поговорим в главах 10 и 15. А пока загрузим модуль. Начнем с модуля File::Basename, поскольку это один из базовых модулей. Чтобы загрузить его, в нашем сценарии мы должны написать:

```
use File::Basename;
```

После этого мы получим возможность обращаться к подпрограммам fileparse, basename и dirname[2] из нашего сценария[3]. Начиная с этого момента мы можем написать

```
my $basename = basename( $some_full_path );
my $dirname = dirname( $some_full_path );
```

Точно так же, как если бы мы сами написали подпрограммы basename и dirname или как если бы они были встроенными функциями Perl. Эти подпрограммы возвращают имя файла и имя каталога, выбирая их из полного пути к файлу. Допустим, что в переменной $some_full_ path содержится строка D:\Projects\IslandRescue\plan7.rtf (если исходить из предположения, что программа работает в операционной системе Windows), тогда в переменную $basename будет записано имя файла plan7.rtf, а в переменную $dirname – имя каталога D:\Projects\IslandRescue.

Модуль File::Basename автоматически определяет тип операционной системы, благодаря чему его функции корректно обрабатывают символы-разделители, которые могут встретиться в строке.

Однако предположим, что подпрограмма dirname уже есть в программе. Вместо нее будет вызываться одноименная подпрограмма, определение которой находится в модуле File::Basename! Включив режим вывода предупреждений, можно увидеть сообщения, предупреждающие об этом; в противном случае Perl никак не будет реагировать на подобные ситуации.

---

1 В свою программу (иногда называемую, и авторами тоже, «сценарием» – в случае Perl это одно и то же). – *Примеч. науч. ред.*

2 А также к вспомогательной подпрограмме fileparse_set_ftype.

3 Фактически мы импортируем их в текущий пакет, но время говорить об этом пока еще не пришло.

Как составить список импорта

К счастью, можно явно определить, что именно надо импортировать, поместив в директиве `use` вслед за именем модуля список подпрограмм, который называется *списком импорта*:

```
use File::Basename ('fileparse', 'basename');
```

Теперь мы сможем взять только эти две подпрограммы из модуля и предусмотреть собственную реализацию подпрограммы `dirname`. Конечно, такой способ записи кажется громоздким, поэтому чаще всего список импорта записывается с применением оператора `qw`:

```
use File::Basename qw( fileparse basename );
```

Фактически, даже если список подпрограмм содержит всего один элемент, программисты все равно предпочитают оператор `qw()`, обеспечивая тем самым единообразие стиля и простоту обслуживания, так как нередко приходится возвращаться к директиве импорта и дополнять список импортируемых подпрограмм позже, а оператор `qw()` сильно облегчает эту работу.

Таким способом мы защитили нашу собственную реализацию подпрограммы `dirname`. Но что делать, если возникнет потребность воспользоваться подпрограммой `dirname` из модуля `File::Basename`? Никаких проблем! Для этого достаточно указать полное имя подпрограммы, уточнив его именем модуля:

```
my $dirname = File::Basename::dirname($some_path);
```

В этом случае нет необходимости дополнять список импорта в директиве `use`. Мы всегда можем обратиться к подпрограмме по полному имени независимо от содержимого списка импорта:[1]

```
my $basename = File::Basename::basename($some_path);
```

В редких случаях (что иногда бывает удобно) мы можем определить пустой список импорта:

```
use File::Basename ( ); # список импорта не определен
my $base = File::Basename::basename($some_path);
```

Пустой список не эквивалентен отсутствию списка. Он означает, что ничего импортировать не надо. Отсутствие списка означает, что импортировать следует то, что определено по умолчанию. Если автор модуля хорошо сделал свою работу, то по умолчанию мы, скорее всего, получим именно то, что нам необходимо.

---

[1] Нет нужды вставлять символ амперсанда перед именем вызываемой функции, поскольку оно будет известно компилятору после того, как он встретит директиву `use`.

Объектно-ориентированные интерфейсы

В противовес импорту подпрограмм из модуля File::Basename приведем другой базовый модуль – с именем File::Spec. Этот модуль предназначен для организации поддержки операций со спецификациями файлов. (Спецификация файла – это, как правило, имя файла или каталога, а если файла с таким именем нет, то и спецификация – это уже не имя файла, не так ли?)

В отличие от модуля File::Basename, модуль File::Spec имеет в первую очередь объектно-ориентированный интерфейс. Для его загрузки тоже применяется директива use:

```
use File::Spec;
```

Однако, поскольку модуль имеет объектно-ориентированный интерфейс,[1] данная директива не импортирует никаких подпрограмм. Вместо этого мы получаем доступ к функциональности модуля через методы класса. Метод catfile объединяет строки, вставляя между ними соответствующий разделитель имен каталогов:

```
my $filespec = File::Spec->catfile( $homedir{gilligan},
        'web_docs', 'photos', 'USS_Minnow.gif' );
```

Таким способом вызывается метод catfile класса File::Spec, который собирает строку пути к файлу с учетом особенностей операционной системы и возвращает строку результата.[2] Аналогичный синтаксис используется для вызова порядка двух десятков других операций, предоставляемых модулем File::Spec.

Модуль File::Spec реализует еще ряд методов, предназначенных для обработки путей к файлам платформонезависимым способом. Более подробно о проблемах переносимости программного кода можно прочитать в документе *perlpotr*.

Типичный объектно-ориентированный модуль Math:BigInt

Не будем расстраиваться по поводу кажущейся «необъектоориентированности» модуля File::Spec, поскольку в нем нет самих объектов,

---

1 Если потребуется функциональный интерфейс, необходимо использовать имя File::Spec::Functions.

2 В операционной системе UNIX эта строка будет выглядеть, например, как /home/gilligan/web_docs/photos/USS_Minnow.gif. В операционной системе Windows в качестве разделителей имен каталогов будут выступать символы обратного слэша (\). Таким образом, данный модуль позволяет гораздо проще писать переносимый программный код, по крайней мере, с точки зрения спецификаций файлов.

а рассмотрим еще один базовый модуль, `Math::BigInt`, предоставляющий операции для работы с большими целыми числами, поддержка которых отсутствует в самом Perl.[1]

```
use Math::BigInt;

my $value = Math::BigInt->new(2); # начинаем с числа 2

$value->bpow(1000);               # найти 2**1000

print $value->bstr( ), "\n";      # вывести результат
```

Как и прежде, мы ничего не импортируем из модуля. Интерфейс модуля целиком построен на применении методов класса, таких как `new`, который создает экземпляры класса, после чего вызываются методы экземпляров, такие как `bpow` и `bstr`.

Единая архивная сеть Perl

Единая архивная сеть Perl (Comprehensive Perl Archive Network – CPAN) содержит результаты труда массы энтузиастов, многие из которых ранее содержали собственные небольшие (или крупные) сайты FTP еще до того, как появилась Всемирная паутина. До 1993 года они координировали свои действия с помощью списка рассылки *perl-packrats*, после чего решили, что дисковое пространство стало достаточно недорогим, чтобы дублировать одну и ту же информацию на всех сайтах сразу, вместо того чтобы специализировать сайты по разным направлениям. Реализация идеи заняла почти год, и наконец Джаркко Хиетаниеми (Jarkko Hietaniemi) объявил о создании финского сайта FTP, получившего название CPAN и ставшего основным, с которого могли получать обновления все остальные зеркала.

Были выполнены некоторые работы по реорганизации и перестройке архивов. Определены места размещения дистрибутивов Perl для систем, отличных от UNIX, сценариев и исходных текстов самого Perl. Однако самой большой и интересной частью CPAN стали модули.

Модули в CPAN организованы в виде дерева символических ссылок, разделенного по категориям, которые ссылаются на фактические каталоги, где собственно и находятся файлы модулей. В разделе модулей есть предметные указатели в формате, который легко анализируется средствами Perl и подобный тому, в каком модуль `Data::Dumper` выдает результаты анализа подробного предметного указателя модулей. Разумеется, предметный указатель генерируется автоматически из баз данных главного сервера с помощью программ, написанных на языке Perl.

---

[1] Строго говоря, собственные возможности Perl ограничиваются возможностями аппаратной архитектуры, на которой он работает. Это одно из немногих мест в Perl, где проявляются особенности аппаратной части.

Собственно обновление информации на зеркалах CPAN производится с помощью еще одной, уже устаревшей программы `mirror.pl`.

Если ранее CPAN размещался на небольшом числе серверов, то теперь он включает в себя более 200 общедоступных архивов в разных уголках Сети, которые постоянно обновляются – как минимум ежедневно, а некоторые из них даже ежечасно. Независимо от того, где мы будем находиться в тот или иной момент времени, мы всегда сможем найти ближайшее зеркало CPAN и загрузить с него самые последние версии пакетов.

Наверное, больше других вам понравится невероятно удобный CPAN Search (*http://search.cpan.org*). С помощью этого веб-сайта можно искать модули, просматривать их описание и содержимое дистрибутивов, ознакомиться с отчетами тестеров и многое-многое другое.

Установка модулей из CPAN

Простой модуль из CPAN устанавливается без особых затей: архив с дистрибутивом модуля загружается и распаковывается, после чего следует перейти в нужный каталог. Мы загружали модуль при помощи утилиты `wget`, но можно взять и другой инструмент.

```
$ wget http://www.cpan.org/.../HTTP-Cookies-Safari-1.10.tar.gz
$ tar -xzf HTTP-Cookies-Safari-1.10.tar.gz
$ cd HTTP-Cookies-Safari-1.10s
```

С этого момента дальше можно двинуться двумя путями (каждый из них мы подробно опишем в главе 16). Если в каталоге будет обнаружен файл `Makefile.PL`, мы запускаем его, чтобы собрать, протестировать и, наконец, установить модуль.

```
$ perl Makefile.PL
$ make
$ make test
$ make install
```

Если у вас недостаточно прав, чтобы установить модуль в каталог, откуда он будет доступен всем пользователям в системе,[1] то можете установить его в свой собственный каталог, добавив аргумент командной строки `PREFIX`:

```
$ perl Makefile.PL PREFIX=/Users/home/Ginger
```

Чтобы Perl мог отыскать модуль в этом каталоге, необходимо добавить его к содержимому переменной окружения `PERL5LIB`. Perl сам добавит эти каталоги в свой список поиска.

---

1 Местоположение этого каталога определяется системным администратором на этапе установки Perl. Имя каталога можно узнать с помощью команды `perl -V`.

```
$ export PERL5LIB=/Users/home/Ginger
```

Кроме того, можно обратиться к директиве lib, чтобы добавить каталог модуля к пути поиска, хотя это будет не совсем правильно, поскольку на других компьютерах, где должна будет работать программа, каталог с модулем может находиться совсем в другом месте.

```
#!/usr/bin/perl
use lib qw(/Users/home/Ginger);
```

Теперь вернемся на шаг назад. Если вместо файла Makefile.PL мы обнаружим файл Build.PL, то сама последовательность выполняемых действий остается прежней. Но установка таких дистрибутивов выполняется с помощью модуля Module::Build. Поскольку сам модуль Module::Build не является базовым модулем Perl,[1] то его надо установить раньше, чем другие модули, установка которых осуществляется с помощью Module::Build.

```
$ perl Build.PL
$ perl Build
$ perl Build test
$ perl Build install
```

Чтобы установить модуль в каталог по выбору пользователя с помощью модуля Module::Build, необходимо определить параметр командной строки --install_base. В этом случае список каталогов, в которых Perl будет искать требуемые модули, определяется точно так же, как и ранее.

```
$ perl Build.PL --install_base /Users/home/Ginger
```

Иногда в составе дистрибутива можно обнаружить оба файла, Makefile.PL и Build.PL. Как поступить в этом случае? Надо выбрать один из них, а какой – решать вам.

Настройка списка каталогов для поиска модулей

Поиск модулей производится в каталогах, которые перечислены в массиве @INC. Директива use исполняется на этапе компиляции, поэтому просмотр массива @INC и поиск подключаемых модулей производятся также на этапе компиляции. Если сейчас мы не разберемся с массивом @INC, позднее это может привести к появлению ошибок, сложных для понимания.

Предположим, что у нас есть свой собственный каталог /home/gilligan/lib, куда мы поместили наш модуль Navigation::SeatOfPants: /home/gilligan/lib/Navigation/SeatOgPants.pm. Когда мы попытаемся подключить этот модуль, Perl не сможет отыскать его.

[1] По крайней мере, пока. Ожидается, что он будет включен в состав дистрибутива Perl начиная с версии 5.10.

```
use Navigation::SeatOfPants;
```

Компилятор Perl сообщит, что не смог отыскать путь к модулю в массиве @INC и покажет все каталоги, которые перечислены в этом массиве.

```
Can't locate Navigation/SeatofPants.pm in @INC (@INC contains: ...)
```

На первый взгляд может показаться, что ситуацию удастся разрешить простым добавлением каталога в массив @INC перед обращением к директиве use. Однако даже такой программный код:

```
unshift @INC, '/home/gilligan/lib'; # не решает проблему
use Navigation::SeatOfPants;
```

нам не поможет. Почему? Потому что unshift выполняется на этапе исполнения – уже после того, как на этапе компиляции будет обработана директива use. Лексически эти две директивы расположены рядом и следуют одна за другой, но исполняются в разное время. Порядок записи отнюдь не всегда совпадает с порядком исполнения. Итак, требуется изменить содержимое массива @INC до того, как будет вызвана директива use. Один из способов состоит в том, чтобы заключить вызов unshift в блок BEGIN:

```
BEGIN { unshift @INC, '/home/gilligan/lib'; }
use Navigation::SeatOfPants;
```

Теперь содержимое блока BEGIN будет исполняться на этапе компиляции, благодаря чему Perl сможет отыскать модуль, подключаемый в директиве use.

Но такой способ не очень удобен, и вам потребуется написать подробный комментарий, чтобы объяснить все тонкости программисту, который должен будет сопровождать и модифицировать ваш программный код. Попробуем добиться того же эффекта более простым способом, который мы уже применяли:

```
use lib '/home/gilligan/lib';
use Navigation::SeatOfPants;
```

Директива lib принимает один или более аргументов и добавляет их в начало массива @INC точно так же, как и использованный нами оператор unshift.[1] Этот прием срабатывает потому, что данная директива исполняется на этапе компиляции. Поэтому порядок исполнения директив соответствует порядку их следования в исходных текстах.

Аргументы директивы use lib практически всегда зависят от местоположения модулей в системе, поэтому стало уже традицией (которой мы призываем вас следовать) помещать эти директивы в самое начало файла. Благодаря этому их нетрудно найти и изменить в случае необходимо-

[1] Кроме того, директива use lib не сдвигает архитектурнозависимые библиотеки вниз относительно запрашиваемой, что делает ее более привлекательной по сравнению со способом, представленным ранее.

сти, например при переносе программного обеспечения в другую систему или при изменении имени каталога с модулями. (Разумеется, мы можем вообще отказаться от директивы use lib, для чего достаточно установить модуль в один из предопределенных каталогов, которые перечислены в массиве @INC по умолчанию, но это возможно далеко не всегда.)

Директива use lib должна интерпретироваться не как «use this lib» (подключить эту библиотеку), а как «use this path to find my libraries (and modules)» (добавить этот каталог в список поиска библиотек (и модулей)). Очень часто программисты допускают такую ошибку:

```
use lib '/home/gilligan/lib/Navigation/SeatOfPants.pm'; # НЕВЕРНО!
```

После чего они долго думают, почему Perl не может найти модуль. Не забывайте, директива use lib исполняется на этапе компиляции, поэтому следующий вариант тоже не сработает:

```
my $LIB_DIR = '/home/gilligan/lib';
...
use lib $LIB_DIR; # ОШИБКА!
use Navigation::SeatOfPants;
```

Конечно же, Perl создаст переменную $LIB_DIR на этапе компиляции (по этой причине мы не получим сообщение о некорректном использовании переменной, хотя сообщение о невозможности отыскать модуль все-таки будет выведено), но значение этой переменной будет присвоено на этапе исполнения. Опять слишком поздно!

Чтобы этот прием был эффективным, необходимо вставить операцию присваивания в блок BEGIN или определить переменную как константу с помощью директивы constant:

```
use constant LIB_DIR => '/home/gilligan/lib';
...
use lib LIB_DIR;
use Navigation::SeatOfPants;
```

Ошибку снова удалось исправить! Но это возможно лишь до тех пор, пока путь к каталогу с требуемой библиотекой известен заранее и для его определения не требуются сложные вычисления. (Когда же это закончится? Кто-то должен положить конец этому безумию!) Хотя в 99 случаях из 100 этого бывает вполне достаточно.

Обработка зависимостей модулей

Мы только что говорили, что для установки некоторых модулей необходимо сначала установить модуль Module::Build. Это легкая форма мигрени под названием «Зависимости между модулями», и даже все кокосовые орехи на острове, куда попали наши герои, потерпевшие кораблекрушение, не облегчат ее. Нам может понадобиться установить всего несколько модулей, однако каждый из них может потребо-

вать установки дополнительных модулей для удовлетворения всех зависимостей.

К счастью, имеется инструмент, способный помочь в этом. Модуль CPAN.pm вошел в состав базовых модулей начиная с версии Perl 5.004. Он реализует диалоговую оболочку установки модулей.

```
$ perl -MCPAN -e shell
cpan shell -- CPAN exploration and modules installation (v1.7601)
ReadLine support available (try 'install Bundle::CPAN')

cpan>
```

Чтобы установить модуль со всеми его зависимостями, достаточно ввести команду install с именем требуемого модуля. После этого CPAN.pm сам загрузит, распакует, скомпилирует, протестирует и установит требуемый модуль, рекурсивно удовлетворяя все обнаруженные зависимости.

```
cpan> install CGI::Prototype
```

Такой подход не требует от нас больших усилий, однако Брайан [Д. Фой] пошел еще дальше и написал сценарий cpan, который также включен в состав Perl. Этому сценарию достаточно передать список модулей, а все остальное он сделает сам.

```
$ cpan CGI::Prototype HTTP::Cookies::Safari Test::Pod
```

Еще один инструмент, CPANPLUS, представляет собой полностью переписанный модуль CPAN.pm, но он не относится к базовым.

```
$ perl -MCPANPLUS -e shell
CPANPLUS::Shell::Default -- CPAN exploration and modules installation (v0.03)
*** Please report bugs to <cpanplus-bugs@lists.sourceforge.net>.
*** Using CPANPLUS::Backend v0.049.
*** ReadLine support available (try 'i Term::ReadLine::Perl').

CPAN Terminal>
```

Для установки модуля предназначена команда i.

```
CPAN Terminal> i CGI::Prototype
```

В состав модуля CPANPLUS входит вспомогательный сценарий cpanp, аналогичный упоминавшемуся ранее сценарию cpan. Чтобы установить группу модулей с его помощью, достаточно передать сценарию команду i и список требуемых модулей.

```
$ cpanp i CGI::Prototype HTTP::Cookies::Safari Test::Pod
```

Упражнения

Ответы на эти вопросы вы найдете в приложении A в разделе «Ответы к главе 3».

Упражнение 1 [25 мин]

Прочитайте список файлов в текущем каталоге и преобразуйте полученные имена в их спецификации, содержащие полные пути к файлам. При этом нельзя обращаться к возможностям командной оболочки или каких-либо внешних программ. Допускается использовать модули `File::Spec` и `Cwd`, входящие в состав Perl. При выводе списка спецификаций перед каждой из них должны выводиться четыре пробела, а после каждой из них – символ перевода строки (точно так же, как это делалось в упражнении 1 из главы 2). Сможете ли вы задействовать программный код, написанный ранее?

Упражнение 2 [35 мин]

Попробуйте выполнить интерпретацию ISBN этой книги, который вы найдете на последней странице обложки (0596102062).[1] Установите модуль `Business::ISBN` из CPAN и воспользуйтесь им для извлечения кода страны и кода издателя из указанного числа.

[1] Это, понятное дело, ISBN английского издания.

4

Введение в ссылки

Ссылки – это основа сложных структур данных, объектно-ориентированного программирования (ООП) и необычной работы с подпрограммами. Ссылки были добавлены в Perl между версиями 4 и 5.

Скалярные переменные Perl могут хранить одиночное значение. Массив представляет собою упорядоченный список из одного или более скалярных значений. Хеши могут хранить коллекции скаляров – одни в виде ключей, другие в виде значений. Хотя скаляр и может быть любой строкой, в которую можно «втиснуть» массив или хеш, ни один из этих трех типов данных не может использоваться для описания сложных взаимосвязей между данными. Для этого лучше всего подходят ссылки. Чтобы прочувствовать, насколько важны ссылки, начнем с одного примера.

Выполнение однотипных действий с разными массивами

Прежде чем «Minnow» (Пескарь) сможет отплыть от пристани (например, на трехчасовую экскурсию), мы должны проверить всех пассажиров и членов экипажа на наличие у них всего необходимого. Скажем, для экскурсии по морю каждый, кто находится на борту, должен иметь солнечные очки, крем или лосьон от загара, фляжку с водой и накидку от дождя. Чтобы проверить, как подготовился к этому мероприятию Шкипер, достаточно написать совсем немного программного кода:

```
my @required = qw(солнечные_очки крем фляжка_с_водой накидка);
my @skipper = qw(голубая_рубашка шляпа накидка солнечные_очки крем);

for my $item (@required) {
    unless (grep $item eq $_, @skipper) { # нет в списке?
```

```
        print "Шкипер: отсутствует $item.\n";
    }
}
```

В скалярном контексте grep возвращает количество раз, когда в результате вычисления выражения $item eq $_ было получено значение «истина» (1 – это истина, 0 – ложь).[1] Если получилось значение 0 (то есть «ложь»), мы выводим сообщение.

Разумеется, чтобы проверить экипировку Джиллигана или Профессора, нам придется написать:

```
my @gilligan = qw(красная_рубашка шляпа счастливые_носки фляжка_с_водой);
for my $item (@required) {
    unless (grep $item eq $_, @gilligan) { # нет в списке?
        print "Джиллиган: отсутствует $item.\n";
    }
}
my @professor = qw(крем фляжка_с_водой рулетка батарейки радиоприемник);
for my $item (@required) {
    unless (grep $item eq $_, @professor) { # нет в списке?
        print "Профессор: отсутствует $item.\n";
    }
}
```

Вы наверняка заметили, что в этом примере постоянно повторяется слишком большой объем программного кода. Неплохо было бы выделить повторяющиеся строки в виде отдельной подпрограммы (и вы совершенно правы):

```
sub check_required_items {
    my $who = shift;
    my @required = qw(солнечные_очки крем фляжка_с_водой накидка);
    for my $item (@required) {
        unless (grep $item eq $_, @_) { # нет в списке?
            print "$who: отсутствует $item.\n";
        }
    }
}

my @gilligan = qw(красная_рубашка шляпа счастливые_носки фляжка_с_водой);
check_required_items('Джиллиган', @gilligan);
```

Изначально подпрограмме передаются 5 аргументов в виде массива @_: имя Джиллиган и четыре предмета, принадлежащие Джиллигану. После выполнения функции shift в массиве @_ останутся только предметы. Затем grep будет вынимать предметы из контрольного списка и проверять их наличие в списке имеющихся предметов.

[1] Есть более эффективный способ проверки наличия элемента в очень больших списках. Но для нескольких элементов приведенный способ, пожалуй, самый простой.

Пока все идет неплохо. Аналогичным образом мы можем проверить Шкипера и Профессора, добавив всего несколько строк:

```
my @skipper = qw(голубая_рубашка шляпа накидка солнечные_очки крем);
my @professor = qw(крем фляжка_с_водой рулетка батарейки радиоприемник);
check_required_items('Шкипер', @skipper);
check_required_items('Профессор', @professor);
```

Для проверки других пассажиров корабля придется повторить те же действия. Этот программный код отвечает первоначальным требованиям, но в нем имеются две неувязки:

- Чтобы создать массив @_, Perl вынужден копировать все содержимое проверяемого массива. Это не очень важно, если объем массива невелик, но если массив большой, то копировать его элементы только ради того, чтобы передать их в подпрограмму, слишком расточительно.
- Предположим, что требуется изменить оригинальный массив, чтобы, например, включить в него обязательные элементы. Поскольку подпрограмме передается копия массива («передача аргумента по значению»), все изменения, произведенные в массиве @_, будут потеряны после выхода из подпрограммы.[1]

Любую из этих неувязок можно устранить, передав массив не по значению, а по ссылке. И это как раз то, что доктор (то есть Профессор) прописал.

Ссылки на массивы

Кроме всего прочего, символом обратного слэша (\) обозначается оператор «взятия ссылки». Когда этот оператор ставится перед именем массива, например \@skipper, мы получаем *ссылку* на этот массив. Ссылка на массив сродни указателю: она указывает на массив, но сама не является массивом.

Ссылка может применяться там же, где и скалярные величины. Ссылки могут указывать на элементы массива или хеша или на обычные скалярные переменные, например так:

```
my $reference_to_skipper = \@skipper;
```

Ссылки можно копировать:

```
my $second_reference_to_skipper = $reference_to_skipper;
```

или даже:

```
my $third_reference_to_skipper = \@skipper;
```

---

1 На самом деле, после того как shift изменит соответствующую переменную, скалярным элементам массива @_ можно присвоить новые значения, но это по-прежнему не дает нам возможности расширять массив дополнительными элементами.

Мы можем оперировать всеми тремя ссылками и даже утверждать, что они идентичны, поскольку ссылаются на один и тот же массив.

```
if ($reference_to_skipper == $second_reference_to_skipper) {
    print "Эти ссылки идентичны.\n";
}
```

В данном случае сравниваются числовые представления двух ссылок. Числовое представление – это уникальный адрес в памяти массива `@skipper`, структура которого не меняется в течение всего времени жизни переменной. Если попробовать посмотреть на строковое представление ссылки (с помощью оператора `eq` или `print`), мы увидим строку:

```
ARRAY(0x1a2b3c)
```

которая содержит уникальный адрес массива (о чем говорит шестнадцатеричная форма представления). Кроме того, данная строка свидетельствует, что это ссылка на массив. Разумеется, если нечто подобное появится в выводе нашей программы, то почти наверняка будет означать ошибку, поскольку дампы шестнадцатеричных данных не представляют интереса для пользователей!

Ссылки допускают копирование и передачу подпрограммам, и это обстоятельство можно использовать для передачи нашей подпрограмме ссылки на массив:

```
my @skipper = qw(голубая_рубашка шляпа накидка солнечные_очки крем);
check_required_items("Шкипер", \@skipper);

sub check_required_items {
    my $who = shift;
    my $items = shift;
    my @required = qw(солнечные_очки крем фляжка_с_водой накидка);
    ...
}
```

Теперь переменная `$items` внутри подпрограммы представляет собой ссылку на массив `@skipper`. Но как добраться до самого массива, имея ссылку на него? Для этого достаточно *разыменовать* ссылку.

Разыменование ссылок на массивы

Если внимательно посмотреть на имя массива `@skipper`, можно заметить, что оно состоит из двух частей: символа `@` и собственно имени массива. Точно так же запись `$skipper[1]` синтаксически состоит из имени массива и дополнительной синтаксической конструкции, которая указывает на второй элемент массива (индекс массива со значением 1 соответствует второму элементу массива, поскольку первый элемент имеет индекс 0).

Хитрость заключается в том, чтобы заключить ссылку в фигурные скобки. Таким образом, везде, где необходимо обратиться к массиву, доста-

точно написать имя ссылки, заключенное в фигурные скобки: { $items }. Например, следующие две строки ссылаются на весь массив:

```
@ skipper
@{ $items }
```

а следующие две – на второй элемент массива:

```
$ skipper [1]
${ $items }[1]
```

Посредством ссылок на массивы нам удалось отделить программный код обращения к массиву от самого массива. Посмотрим, как это повлияло на оставшуюся часть подпрограммы:

```
sub check_required_items {
    my $who = shift;
    my $items = shift;
    my @required = qw(солнечные_очки крем фляжка_с_водой накидка);

    for my $item (@required) {
        unless (grep $item eq $_, @{$items}) { # нет в списке?
            print "$who: отсутствует $item.\n";
        }
    }
}
```

Мы всего лишь заменили @_ (копию проверяемого списка) на @{$items}, то есть разыменовали ссылку на оригинальный массив. Теперь, как и прежде, мы можем вызвать подпрограмму несколько раз:

```
my @skipper = qw(голубая_рубашка шляпа накидка солнечные_очки крем);
check_required_items('Шкипер', \@skipper);

my @professor = qw(крем фляжка_с_водой рулетка батарейки радиоприемник);
check_required_items('Профессор', \@professor);

my @gilligan = qw(красная_рубашка шляпа счастливые_носки фляжка_с_водой);
check_required_items('Джиллиган', \@gilligan);
```

Каждый раз $items будет указывать на разные массивы, что позволяет выполнять проверки посредством одного и того же программного кода. Этот пример демонстрирует один из самых важных случаев применения ссылок: отделение логики программы от структур данных позволяет многократно задействовать один и тот же программный код.

Передача массива в подпрограмму по ссылке устраняет первую из двух неувязок, о которых мы говорили выше. Теперь, вместо того чтобы целиком копировать исходный массив в массив @_, мы передаем подпрограмме единственный элемент – ссылку на массив.

Можно ли убрать два вызова shift в начале подпрограммы? Конечно, но для этого придется пожертвовать ясностью кода подпрограммы:

```
sub check_required_items {
    my @required = qw(солнечные_очки крем фляжка_с_водой накидка);
```

```
        for my $item (@required) {
            unless (grep $item eq $_, @{$_[1]}) { # нет в списке?
                print "$_[0]: отсутствует $item.\n";
            }
        }
    }
```

У нас в массиве @_ по-прежнему два элемента. Первый – это имя пассажира или члена экипажа, которое фигурирует в выводе сообщения об ошибке. Второй элемент – это ссылка на массив с экипировкой, который передается оператору grep.

Избавляемся от фигурных скобок

В большинстве случаев ссылки служат для обеспечения доступа к скалярным величинам, таким как @{$items} или ${$items}[1]. В таких ситуациях фигурные скобки можно опустить и записывать ссылки так: @$items или $$items[1].

Однако избавиться от фигурных скобок невозможно, если значение в скобках не является скалярной величиной. Например в такой форме записи: @{$_[1]}, которую мы использовали в последней версии подпрограммы. Здесь мы не можем избавиться от фигурных скобок, поскольку в данном случае элемент массива представляет собой массив, а не скалярную величину.

В соответствии с этим правилом нетрудно определить, где должны стоять «отсутствующие» фигурные скобки. Так, если мы встречаем запись в виде $$items[1], то можно с уверенностью утверждать, что фигурные скобки должны обрамлять простую скалярную переменную $items. Отсюда следует вывод, что $items представляет собой ссылку на массив.

Таким образом, самая простая для восприятия человеком версия подпрограммы может выглядеть следующим образом:

```
sub check_required_items {
    my $who = shift;
    my $items = shift;

    my @required = qw(солнечные_очки крем фляжка_с_водой накидка);
    for my $item (@required) {
        unless (grep $item eq $_, @$items) { # нет в списке?
            print "$who: отсутствует $item.\n";
        }
    }
}
```

Эта версия отличается от предыдущей лишь отсутствием фигурных скобок.

Модификация массивов

Теперь вы знаете, как с помощью ссылок избавиться от ненужного копирования информации. Посмотрим, как внести изменения в оригинальный массив.

Каждый отсутствующий элемент экипировки мы будем добавлять в массив, обращая на него внимание пассажира:

```
sub check_required_items {
    my $who = shift;
    my $items = shift;

    my @required = qw(солнечные_очки крем фляжка_с_водой накидка);
    my @missing = ( );

    for my $item (@required) {
        unless (grep $item eq $_, @$items) { # нет в списке?
            print "$who: отсутствует $item.\n";
            push @missing, $item;
        }
    }

    if (@missing) {
        print "Добавлены @missing в саквояж @$items пассажира $who.\n";
        push @$items, @missing;
    }
}
```

Обратите внимание на дополнительный массив `@missing`. Если в процессе проверки багажа пассажира обнаруживается отсутствующие обязательные элементы экипировки, мы записываем их в массив `@missing`. Если по окончании проверки массив не пустой, мы добавляем его содержимое к оригинальному массиву с экипировкой.

Ключевой является последняя строка подпрограммы. Здесь мы разыменовали ссылку на оригинальный массив `$items` и добавили в него содержимое массива `@missing`. При отсутствии ссылки на массив мы смогли бы изменить только локальную копию данных, что никак не повлияло бы на сам массив.

Кроме того, запись `@$items` (и более универсальная форма `@{$items}`) может применяться внутри строк, заключенных в двойные кавычки. Между символом `@` и остальной частью ссылки нельзя вставлять пробелы, зато внутри фигурных скобок они вполне допустимы, как в обычном программном коде Perl.

Вложенные структуры данных

В данном примере массив `@_` содержит два элемента, один из которых сам является массивом. А как быть, если мы получаем ссылку на массив, который сам содержит ссылки на другие массивы? Такая сложная организация данных может оказаться весьма удобной.

Например, мы можем обойти массивы Шкипера, Джиллигана и Профессора и построить один большой список предметов экипировки, имеющихся в наличии:

```
my @skipper = qw(голубая_рубашка шляпа накидка солнечные_очки крем);
my @skipper_with_name = ('Шкипер', \@skipper);
my @professor = qw(крем фляжка_с_водой рулетка батарейки радиоприемник);
my @professor_with_name = ('Профессор', \@professor);
my @gilligan = qw(красная_рубашка шляпа счастливые_носки фляжка_с_водой);
my @gilligan_with_name = ('Джиллиган', \@gilligan);
```

В данном случае массив `@skipper_with_name` содержит два элемента, второй из которых представляет собой ссылку на массив, аналогичную той, что мы передавали в подпрограмму. Теперь можно объединить всю информацию в единый список:

```
my @all_with_names = (
    \@skipper_with_name,
    \@professor_with_name,
    \@gilligan_with_name,
);
```

Обратите внимание: мы получили три элемента, каждый из которых является ссылкой на массив, состоящий из двух элементов: имени и списка имеющихся в наличии предметов. Изображение этой структуры приводится на рис. 4.1.

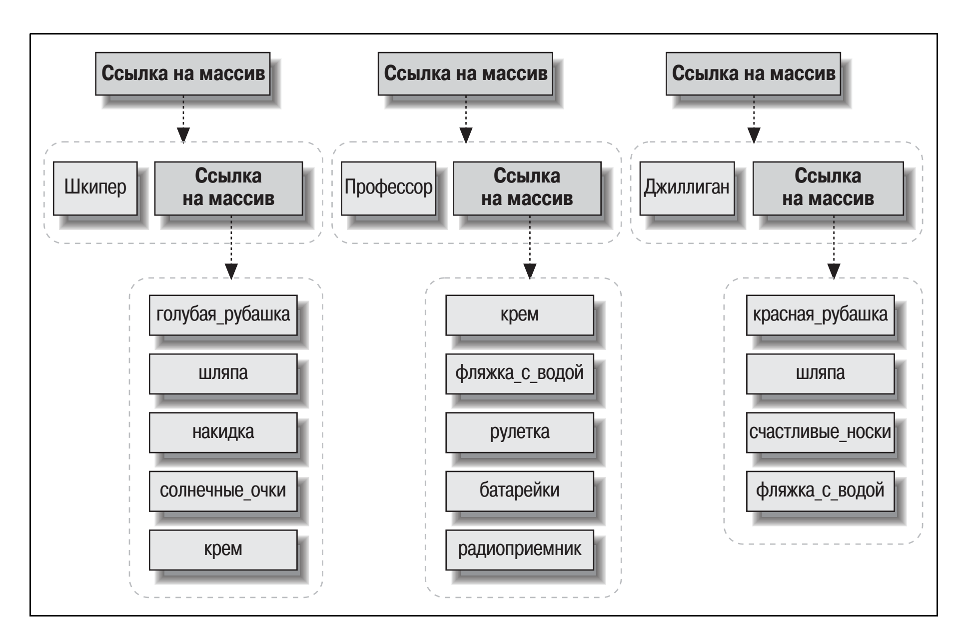


Рис. 4.1. *В массиве @all_with_names сосредоточена вся многоуровневая структура данных, состоящая из строк и ссылок на массивы*

Таким образом, $all_with_names[2] представляет собой ссылку на массив с информацией о Джиллигане. Если разыменовать ссылку как @{$all_with_names[2]}, получится массив из двух элементов: «Джиллиган» и ссылка на другой массив.

Как организовать доступ к данным по этой ссылке? Опять применяя наше правило: ${$all_with_names[2]}[1]. Или, говоря другими словами, берем ссылку $all_with_names[2], разыменовываем ее и обращаемся как к обычному массиву, например $DUMMY[1], где DUMMY заменяется на $all_with_names[2].

А как тогда обращаться к подпрограмме check_required_items(), имея такую структуру данных? Ниже приводится достаточно простой пример:

```
for my $person (@all_with_names) {
    my $who = $$person[0];
    my $provisions_reference = $$person[1];
    check_required_items($who, $provisions_reference);
}
```

Для этого не надо вносить изменения в саму подпрограмму. В процессе исполнения цикла переменная $person будет поочередно получать значения $all_with_names[0], $all_with_names[1] и $all_with_names[2]. Когда разыменовывается ссылка $$person[0], мы поочередно будем получать строки «Шкипер», «Профессор» и «Джиллиган». Соответственно разыменованная ссылка $$person[1] будет представлять список вещей, принадлежащих этим персонажам.

Разумеется, мы можем сократить объем кода, отказавшись от промежуточных переменных:

```
for my $person (@all_with_names) {
    check_required_items(@$person);
}
```

или даже так:

```
check_required_items(@$_) for @all_with_names;
```

Как видите, различные уровни оптимизации могут приводить к снижению ясности программного кода. Представьте себе, что может вам прийти в голову, когда через месяц вы попытаетесь разобраться в своем собственном программном коде. Если этот довод кажется вам неубедительным, представьте себе, каково будет другому программисту, который пожелает подхватить ваши начинания.[1]

1 В издательстве O'Reilly вышла прекрасная книга «Perl Best Practicies» Дэмиана Конвея (Damian Conway), в которой приводятся 256 советов, как сделать программный код на языке Perl удобочитаемым и более простым в поддержке.

Упрощаем доступ к вложенным структурам с помощью стрелок

Взглянем еще раз на форму разыменования ссылок с помощью фигурных скобок. Согласно нашему предыдущему примеру ссылка на массив с предметами, принадлежащими Джиллигану, записывается как `${$all_with_names[2]}[1]`. Как теперь получить название первого предмета из списка? Для этого надо разыменовать данный элемент еще раз, то есть добавить еще одни фигурные скобки: `${${$all_with_names[2]}[1]}[0]`. Очень неудобная форма записи. А есть ли способ попроще? Конечно есть!

Форму записи `${DUMMY}[$y]` во всех случаях можно заменить более краткой формой `DUMMY->[$y]`. Другими словами, можно разыменовать ссылку на массив, указав в квадратных скобках требуемый элемент массива после выражения, определяющего ссылку на массив, и стрелки.

В нашем примере ссылку на массив с предметами, принадлежащими Джиллигану, можно получить так: `$all_with_names[2]->[1]`, а так – доступ к первому элементу этого массива: `$all_with_names[2]->[1]->[0]`. Вот это да! Такая форма записи воспринимается гораздо проще!

Есть и еще одно правило (как будто *эта форма* еще недостаточно проста): если стрелка встречается между элементами, обозначающими индекс, к примеру между квадратными скобками, то ее можно опустить: `$all_with_names[2]->[1]->[0]` превращается в `$all_with_names[2][1][0]`. Эта форма более удобочитаема.

Оператор стрелки обязательно должен присутствовать после имени ссылки, а между индексами его можно опустить. Почему? Представьте себе следующую ссылку на массив `@all_with_names`:

```
my $root = \@all_with_names;
```

Как теперь получить название первого предмета, принадлежащего Джиллигану?

```
$root -> [2] -> [1] -> [0]
```

В соответствии с правилом, определенным выше, стрелки между индексами можно опустить:

```
$root -> [2][1][0]
```

Но мы не можем опустить оператор стрелки, следующей за именем ссылки, потому что такая форма записи будет означать обращение к третьему элементу массива `$root`, то есть обращение к данным с совершенно иной структурой. Теперь сравним эту форму записи с универсальной, где есть фигурные скобки:

```
${${${$root}[2]}[1]}[0]
```

Форма записи со стрелкой выглядит намного лучше. Однако чтобы получить по ссылке весь массив с информацией о Джиллигане, нам пона-

добится форма записи с фигурными скобками. Например, чтобы получить весь массив с предметами, мы должны записать:

```
@{$root->[2][1]}
```

Такая форма записи должна читаться изнутри наружу:

- Берем ссылку `$root`.
- Разыменовываем ее как ссылку на массив и берем третий элемент массива (элемент с индексом 2).
- Разыменовываем его как ссылку на массив и берем второй элемент массива (элемент с индексом 1).
- Разыменовываем его как ссылку на массив и берем весь массив целиком.

Последний шаг не имеет краткой формы записи. Ну и ладно.[1]

Ссылки на хеши

Аналогично тому, как можно получить ссылки на массивы, точно так же можно получить ссылки на хеши. Здесь тоже используется символ обратного слэша (\), который является оператором «взятия ссылки»:

```
my %gilligan_info = (
    name => 'Джиллиган',
    hat => 'Белый',
    shirt => 'Красный',
    position => 'Первый помощник капитана',
);
my $hash_ref = \%gilligan_info;
```

Чтобы извлечь информацию о Джиллигане с помощью ссылки, мы должны разыменовать ее. Стратегия работы со ссылками на хеши похожа на стратегию работы со ссылками на массивы. Сначала мы записываем обращение к ссылке, как если бы это был сам хеш, а затем добавляем пару фигурных скобок. Например, выбрать значение конкретного ключа можно так:

```
my $name = $ gilligan_info { 'name' };
my $name = $ { $hash_ref } { 'name' };
```

В этом случае фигурные скобки имеют двойное назначение. Первая пара скобок обозначает выражение, которое возвращает ссылку, а вторая определяет ключ, значение которого должно быть получено.

Для выполнения действий над всем хешем можно записать:

```
my @keys = keys % gilligan_info;
my @keys = keys % { $hash_ref };
```

---

[1] Эта проблема неоднократно обсуждалась разработчиками Perl, но никто из них не смог придумать синтаксис, который был бы обратно совместим с универсальной формой записи.

Как и в случае ссылок на массивы, при некоторых обстоятельствах мы можем использовать сокращенную форму записи, опустив фигурные скобки. Например, если элемент в фигурных скобках является скалярной величиной (как показано в этих примерах), фигурные скобки можно опустить:

```
my $name = $$hash_ref{'name'};
my @keys = keys %$hash_ref;
```

Как и в случае с массивами, при обращении к конкретным элементам хеша допускается применять оператор стрелки:

```
my $name = $hash_ref->{'name'};
```

Поскольку ссылки допускаются везде, где допускаются скалярные переменные, мы можем создать массив ссылок на хеши:

```
my %gilligan_info = (
    name => 'Джиллиган',
    hat => 'Белый',
    shirt => 'Красный',
    position => 'Первый помощник капитана',
);
my %skipper_info = (
    name => 'Шкипер',
    hat => 'Черный',
    shirt => 'Голубой',
    position => 'Капитан',
);
my @crew = (\%gilligan_info, \%skipper_info);
```

Здесь элемент `$crew[0]` представляет ссылку на хеш с информацией о Джиллигане. Имя Джиллигана в этом случае можно получить любым из следующих способов:

```
${ $crew[0] } { 'name' }
my $ref = $crew[0]; $$ref{'name'}
$crew[0]->{'name'}
$crew[0]{'name'}
```

В последней строке мы просто опустили оператор стрелки «между индексами», хотя левая часть выражения — это массив, а правая — элемент хеша.

Попробуем вывести список членов экипажа:

```
my %gilligan_info = (
    name => 'Джиллиган',
    hat => 'Белый',
    shirt => 'Красный',
    position => 'Первый помощник капитана',
);
my %skipper_info = (
    name => 'Шкипер',
```

```
    hat => 'Черный',
    shirt => 'Голубой',
    position => 'Капитан',
);
my @crew = (\%gilligan_info, \%skipper_info);

my $format = "%-15s %-7s %-7s %-15s\n";
printf $format, qw(Name Shirt Hat Position);
for my $crewmember (@crew) {
    printf $format,
        $crewmember->{'name'},
        $crewmember->{'shirt'},
        $crewmember->{'hat'},
        $crewmember->{'position'};
}
```

Последняя часть выглядит не очень красиво из-за повторяющихся элементов. Мы можем сократить объем кода, использовав синтаксис обращения к части хеша. Универсальная форма записи имеет следующий вид:

```
@gilligan_info { qw(name position) }
```

Обращение к хешу по ссылке будет выглядеть следующим образом:

```
@{ $hash_ref } { qw(name position) }
```

Мы можем отбросить первую пару фигурных скобок, поскольку внутри них находится обычная скалярная величина:

```
@$hash_ref { qw(name position) }
```

Теперь заключительный цикл можно записать в следующей форме:

```
for my $crewmember (@crew) {
    printf $format, @$crewmember{qw(name shirt hat position)};
}
```

Для доступа к части массива или хеша по ссылке с помощью оператора стрелки (->) краткая форма записи отсутствует, точно так же как она отсутствует для доступа ко всему массиву или хешу.

При выводе ссылки на хеш она будет выглядеть как HASH(0x1a2b3c), демонстрируя шестнадцатеричный адрес хеша в памяти. Эта информация совершенно бессмысленна для пользователя и может пригодиться только программисту как признак отсутствия операции разыменования.

Упражнения

Ответы на эти вопросы вы найдете в приложении A в разделе «Ответы к главе 4» .

Упражнение 1 [5 мин]

К скольким различным элементам обращаются следующие строки?

```
$ginger->[2][1]
${$ginger[2]}[1]
$ginger->[2]->[1]
${$ginger->[2]}[1]
```

Упражнение 2 [30 мин]

Опираясь на последнюю версию подпрограммы `check_required_items`, напишите подпрограмму `check_items_for_all`, которая принимала бы ссылку на хеш в качестве единственного аргумента и выводила бы список пассажиров, взошедших на борт «Minnow», и список предметов, поднятых ими на борт.

Хеш ссылок можно сконструировать примерно следующим образом:

```
my @gilligan = ... вещи, принадлежащие Джиллигану ...;
my @skipper = ... вещи, принадлежащие Шкиперу ...;
my @professor = ... вещи, принадлежащие Профессору ...;
my %all = (
    Gilligan => \@gilligan,
    Skipper => \@skipper,
    Professor => \@professor,
);
check_items_for_all(\%all);
```

Созданная подпрограмма должна обращаться к `check_required_items` для проверки каждого из пассажиров и членов экипажа и дополнения списка необходимыми предметами.

5

Ссылки и области видимости

Ссылки могут копироваться и передаваться подобно обычным скалярным переменным. В любой момент времени Perl имеет полную информацию о количестве ссылок на каждый элемент данных. Кроме того, Perl допускает возможность создания ссылок на *анонимные структуры данных* (структуры данных, у которых нет собственного имени) и способен даже создавать ссылки автоматически, если это необходимо для выполнения каких-либо операций. Посмотрим, как копируются ссылки и как это влияет на область видимости и распределение памяти.

Несколько ссылок на данные

В главе 4 было показано, как получить ссылку на массив `@skipper` и сохранить ее в скалярной переменной:

```
my @skipper = qw(голубая_рубашка шляпа накидка солнечные_очки крем);
my $reference_to_skipper = \@skipper;
```

После этого ссылку можно скопировать или взять дополнительные ссылки, при этом все они будут ссылаться на один и тот же объект данных и будут полностью взаимозаменяемы:

```
my $second_reference_to_skipper = $reference_to_skipper;
my $third_reference_to_skipper = \@skipper;
```

Начиная с этого момента у нас имеется четыре разных способа обращения к данным, хранящимся в массиве `@skipper`:

```
@skipper
@$reference_to_skipper
@$second_reference_to_skipper
@$third_reference_to_skipper
```

Perl запоминает количество созданных ссылок для каждого элемента данных; этот механизм называется *подсчетом ссылок*. Исходное имя

структуры считается как одна ссылка. Создание каждой новой ссылки (включая операцию копирования ссылки) увеличивает счетчик ссылок на единицу. В данном случае мы получили четыре ссылки на массив с предметами экипировки.

Ссылки можно создавать и удалять по мере надобности. Пока счетчик ссылок не обнулится, Perl будет хранить массив в памяти и предоставлять к нему доступ любым из имеющихся способов. Например, мы могли бы создать временную ссылку:

```
check_provisions_list(\@skipper)
```

В момент вызова подпрограммы Perl создаст пятую ссылку и скопирует ее в массив входных аргументов `@_` подпрограммы. Подпрограмма сама может создать дополнительные копии ссылки, а Perl автоматически нарастит счетчик ссылок. Когда подпрограмма вернет управление, Perl автоматически уничтожит все ссылки, созданные внутри подпрограммы, и количество ссылок опять будет равно четырем.

Ссылки можно уничтожать, для этого достаточно записать в переменную со ссылкой что-либо, что не является ссылкой на массив `@skipper`. Например, можно присвоить значение `undef`:

```
$reference_to_skipper = undef;
```

Или просто выйти за область видимости переменной:

```
my @skipper = ...;
{ # блок программного кода
    ...
    my $ref = \@skipper;
    ...
    ...
} # в этой точке программа выходит из области видимости $ref
```

В частности, если ссылка хранится в локальной переменной подпрограммы, она автоматически уничтожается, когда подпрограмма возвращает управление.

Неважно, изменяем ли мы сами значение переменной со ссылкой или программа покидает область видимости переменной, Perl автоматически уменьшает счетчик ссылок на объект данных.

Утилизация памяти, занимаемой массивом, производится только когда все ссылки (включая и оригинальное имя массива) выйдут за пределы области видимости. То есть Perl освободит память, занимаемую массивом `@skipper`, только когда будут уничтожены все ссылки на массив.

Эта память будет доступна Perl для размещения других данных этой же программы. И вообще Perl не возвращает память операционной системе.

А если это было имя структуры?

Как правило, все ссылки на переменную уничтожаются до того, как сама переменная выйдет за пределы области видимости. А что произойдет, если ссылка на переменную останется, после того как программа покинет область видимости самой переменной? Рассмотрим следующий пример:

```
my $ref;

{
    my @skipper = qw(голубая_рубашка шляпа накидка солнечные_очки крем);
    $ref        = \@skipper;

    print "$ref->[2]\n"; # выведет слово "накидка\n"
}
print "$ref->[2]\n"; # точно так же выведет слово "накидка\n"
```

Сразу же после объявления массива `@skipper` расположена единственная ссылка на список из пяти элементов. После инициализации переменной `$ref` счетчик ссылок увеличивается на единицу и становится равным двум. После того как поток исполнения выйдет за пределы блока, имя переменной `@skipper` станет недоступным и счетчик ссылок уменьшится на единицу. Однако обращение по имени переменной – это лишь один из двух способов доступа к массиву! Таким образом, список из пяти элементов продолжает существовать в памяти и ссылка `$ref` по-прежнему указывает на него.

В этой точке программы список становится *анонимным массивом*, то есть массивом без имени.

До тех пор пока значение переменной `$ref` не изменится или пока программа не выйдет за пределы области видимости этой переменной, нам доступны те же самые возможности по разыменованию ссылки, какие у нас имелись до того, как имя массива вышло за пределы области видимости. Фактически массив остается все тем же массивом со всеми его особенностями. Мы так же можем сокращать или увеличивать число элементов в массиве, как и в любом другом массиве Perl:

```
push @$ref, 'секстан'; # добавить предмет экипировки
print "$ref->[-1]\n"; # выведет слово секстан\n
```

Мы можем даже увеличить счетчик ссылок таким образом:

```
my $copy_of_ref = $ref;
```

или таким, что суть то же самое:

```
my $copy_of_ref = \@$ref;
```

Данные продолжают существовать в памяти до тех пор, пока не будет уничтожена последняя ссылка:

```
$ref = undef; # пока еще нет...
$copy_of_ref = undef; # теперь все!
```

Подсчет ссылок и вложенные структуры данных

Данные продолжают существовать в памяти до тех пор, пока не будет уничтожена последняя ссылка, даже если ссылка на эти данные находится внутри других структур данных. Предположим, что один из элементов массива является ссылкой. Вернемся к примеру из главы 4:

```
my @skipper = qw(голубая_рубашка шляпа накидка солнечные_очки крем);
my @skipper_with_name = ('Шкипер', \@skipper);

my @professor = qw(крем фляжка_с_водой рулетка батарейки радиоприемник);
my @professor_with_name = ('Профессор', \@professor);

my @gilligan = qw(красная_рубашка шляпа счастливые_носки фляжка_с_водой);
my @gilligan_with_name = ('Джиллиган', \@gilligan);

my @all_with_names = (
    \@skipper_with_name,
    \@professor_with_name,
    \@gilligan_with_name,
    );
```

Теперь представьте, что все промежуточные переменные являются частью подпрограммы:

```
my @all_with_names;

sub initialize_provisions_list {
    my @skipper = qw(голубая_рубашка шляпа накидка солнечные_очки крем);
    my @skipper_with_name = ('Шкипер', \@skipper);

    my @professor = qw(крем фляжка_с_водой рулетка батарейки радиоприемник);
    my @professor_with_name = ('Профессор', \@professor);

    my @gilligan = qw(красная_рубашка шляпа счастливые_носки фляжка_с_водой);
    my @gilligan_with_name = ('Джиллиган', \@gilligan);

    @all_with_names = (
        \@skipper_with_name,
        \@professor_with_name,
        \@gilligan_with_name,
    );
}

initialize_provisions_list( );
```

Мы записали в массив `@all_with_names` три ссылки. Внутри подпрограммы определены именованные массивы, содержащие ссылки на другие именованные массивы. В конечном счете ссылки на все данные сосредотачиваются в глобальном массиве `@all_with_names`. Но когда подпрограмма возвращает управление, имена шести массивов оказываются за пределами области видимости. Внутри подпрограммы было создано по дополнительной ссылке на каждый из массивов, что привело к увеличению счетчиков ссылок. По выходе из подпрограммы счетчики ссылок были уменьшены, но они не стали равны нулю, поэтому

данные остались в памяти, хотя доступ к ним теперь возможен только с помощью массива @all_with_names.

Мы можем отказаться от глобальной переменной @all_with_names и возвращать список в виде возвращаемого значения:

```
sub get_provisions_list {
    my @skipper = qw(голубая_рубашка шляпа накидка солнечные_очки крем);
    my @skipper_with_name = ('Шкипер', \@skipper);

    my @professor = qw(крем фляжка_с_водой рулетка батарейки радиоприемник);
    my @professor_with_name = ('Профессор', \@professor);

    my @gilligan = qw(красная_рубашка шляпа счастливые_носки фляжка_с_водой);
    my @gilligan_with_name = ('Джиллиган', \@gilligan);

    return (
        \@skipper_with_name,
        \@professor_with_name,
        \@gilligan_with_name,
    );
}

my @all_with_names = get_provisions_list( );
```

В данном примере подпрограмма создает возвращаемое значение, представляющее собой список из трех элементов, который затем записывается в переменную @all_with_names. Внутри процедуры были выполнены операции взятия ссылок на каждый из именованных массивов, которые стали частью возвращаемого значения, вследствие чего данные продолжат свое существование в памяти.[1,2] Если мы изменим

---

[1] Сравните с тем, как функции в языке C возвращают массивы. Мы должны либо вернуть указатель на статическую область памяти, и как следствие функция становится нереентерабельной, либо выделить память с помощью функции malloc, но тогда вызывающая программа обязана будет явно освободить эту область с помощью функции free. В случае с Perl все необходимые действия он делает самостоятельно. – *Примеч. авторов.*

По стандартам последних лет для C/C++ (C99, например, или ANSI C) функция может возвращать в качестве результата сложную структуру данных (структуру, массив и т. д.), при этом место для результата резервируется в стеке возврата. Картина наблюдается такая же, что и в Perl, а предыдущее утверждение авторов не имеет силы. – *Примеч. науч. ред.*

[2] Реально картина еще чуть сложнее: а) в подпрограмме создаются массивы, и счетчик ссылок на них становится равным 1; б) в промежуточную анонимную (неименованную) структуру для return записываются значения, и счетчик становится равным 2; в) при завершении подпрограммы определения массивов покидают область видимости, и счетчик уменьшается до 1; г) при копировании возвращаемой ссылки в @all_with_names счетчик снова увеличивается до 2; д) после чего уничтожается анонимная промежуточная структура, возвращенная return, и счетчик снова возвращается в 1. – *Примеч. науч. ред.*

или уничтожим ссылки в массиве `@all_with_names`, Perl уменьшит счетчики ссылок на соответствующие массивы. Если при этом они достигнут нуля (как в данном примере), то будут уничтожены и сами массивы. Поскольку массивы, на которые ссылаются элементы массива `@all_with_names`, тоже содержат ссылки (например, ссылка на массив `@skipper`), Perl аналогичным образом уменьшит соответствующие счетчики ссылок. И опять же, если при этом счетчики ссылок обнулятся, память, занимаемая массивом, будет освобождена.

Уничтожение вершины древовидной структуры данных приводит к уничтожению всех данных, содержащихся в этой структуре. Исключение составляют случаи, когда были созданы дополнительные копии ссылок на вложенные элементы. Например, мы можем скопировать ссылку на список предметов экипировки Джиллигана:

```
my $gilligan_stuff = $all_with_names[2][1];
```

Теперь, когда будет уничтожен массив `@all_with_names`, у нас останется одна ссылка на массив, который изначально был создан под именем `@gilligan`, и соответственно в памяти останутся данные, составляющие содержимое этого массива.

Здесь все просто: пока остается хотя бы одна ссылка на данные, остаются и сами данные.

Ошибки при подсчете ссылок

Основное назначение механизма подсчета ссылок в том, чтобы дать возможность управления памятью в течение длительного периода времени. Один из недостатков этого механизма заключается в том, что он не в состоянии корректно выполнять подсчет при наличии *перекрестных ссылок* – когда одна часть общей структуры создает обратную ссылку на другую часть структуры, образуя замкнутую петлю. Предположим, что у нас есть две структуры, причем обе они содержат ссылки друг на друга (рис. 5.1):

```
my @data1 = qw(one won);
my @data2 = qw(two too to);
```

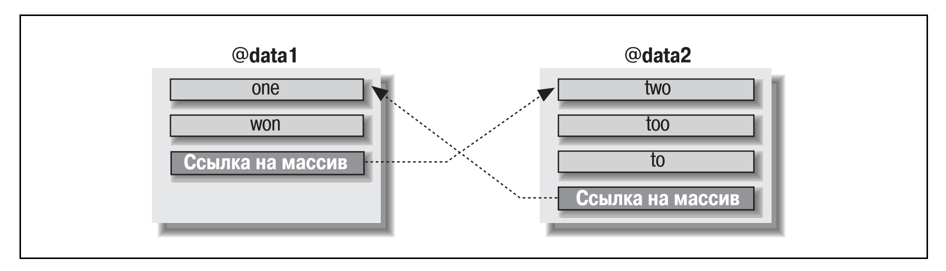


Рис. 5.1. *Когда ссылки на структуры данных образуют замкнутую петлю, механизм подсчета ссылок может оказаться не в состоянии распознать необходимость проведения утилизации областей памяти*

```
push @data2, \@data1;
push @data1, \@data2;
```

Итак, у нас есть два способа обратиться к структуре данных `@data1`: по имени `@data1` и по ссылке `@{$data2[3]}`. Точно так же у нас есть два способа обратиться к структуре данных `@data2`: по имени `@data2` и по ссылке `@{$data2[2]}`. Получился замкнутый круг. Фактически мы получили возможность обратиться к элементу `won` бесконечным числом способов, например `$data1[2][3][2][3][2][3][1]`.

Что произойдет, если оба эти массива выйдут за пределы области их видимости? Прежде всего счетчики ссылок на массивы уменьшатся с двух до одного, но не до нуля. А поскольку счетчики будут иметь ненулевые значения, Perl будет считать, что существует еще какой-то способ обратиться к этим массивам, хотя фактически это не так! Подобные *утечки памяти* приводят к тому, что программа с течением времени начинает потреблять все больше и больше памяти. Тьфу!

Вам может показаться, что данный пример придуман искусственно. Разумеется, в реальной программе мы постараемся избежать создания структур данных, образующих перекрестные ссылки! Но программисты часто создают такие перекрестные ссылки в своих программах, например, для организации двусвязных списков, кольцевых списков и некоторых других структур данных. Ключ к решению этой проблемы заключается в том, что программисты на языке Perl крайне редко прибегают к перекрестным ссылкам, потому что в Perl отсутствуют причины, вынуждающие к этому. В большинстве случаев перекрестные ссылки создаются из-за необходимости управлять памятью и связывать между собой разобщенные блоки памяти. Если вы знакомы с другими языками программирования, то наверняка обратили внимание, что многие задачи программирования на языке Perl могут быть реализованы гораздо проще. Например, насколько просто выполняется сортировка списка элементов, добавление или удаление элементов даже из середины списка. Эти задачи могут оказаться достаточно сложными для реализации в других языках программирования и повлечь необходимость создания перекрестных ссылок между структурами данных, чтобы обойти ограничения, накладываемые языком.[1]

Зачем мы упомянули об этом здесь? Дело в том, что даже программисты, работающие с языком Perl, иногда просто копируют алгоритмы, реализованные на других языках программирования. В этом нет ничего предосудительного, хотя в таких ситуациях лучше будет проанализировать причины, которые заставили автора прибегнуть к перекрестным ссылкам, и затем реализовать тот же алгоритм, но с учетом осо-

[1] Тем не менее это общая и хорошо известная проблема всех систем «сборки мусора» (управления утилизацией неиспользуемой памяти), построенных на простом подсчете числа ссылок к структурам данных. – *Примеч. науч. ред.*

бенностей Perl. По-видимому, здесь можно будет использовать хеш или массив, который впоследствии будет отсортирован.

Вероятно, в будущем вместо или в дополнение к механизму подсчета количества ссылок в Perl появится *сборщик мусора*.[1] Но до тех пор необходимо с большой осторожностью подходить к созданию структур с перекрестными ссылками. И если в этом действительно возникает необходимость, то прежде чем покидать область видимости таких переменных, необходимо в обязательном порядке разрывать образовавшиеся замкнутые петли ссылок, как это сделано, скажем, в следующем примере:

```
{
    my @data1 = qw(one won);
    my @data2 = qw(two too to);
    push @data2, \@data1;
    push @data1, \@data2;
    ... действия над @data1, @data2 ...
    # прежде чем выйти из блока:
    @data1 = ( );
    @data2 = ( );
}
```

Здесь мы принудительно уничтожили ссылки на `@data2` в `@data1` и наоборот. Теперь для каждой структуры имеется только по одной ссылке и по завершении работы блока кода счетчики ссылок обнулятся. На самом деле здесь достаточно было уничтожить ссылку только в одной из структур. В главе 13 показано, как создаются нежесткие ссылки, которые помогут решить многие из указанных проблем.

Создание анонимных массивов

В подпрограмме `get_provisions_list`, о которой говорилось выше, мы создали полдюжины именованных массивов только для того, чтобы получить ссылки на них. По завершении подпрограммы мы выходили из области видимости имен массивов и у нас оставались только ссылки на эти массивы.

Создание временных именованных массивов в простейших случаях не вызывает затруднений, но в случае достаточно сложных структур данных подбирать имена массивов непросто. Нам пришлось бы задумываться об именах массивов лишь для того, чтобы через мгновение забыть о них.

Мы можем ликвидировать затруднения с выбором имен массивов, сузив область видимости имен. Например, вместо того чтобы создавать

[1] Только не спрашивайте нас об этом. Мы писали эту книгу задолго до того, как у нас появилась возможность узнать об этом. Поэтому в то время нам еще ничего не было известно.

массив в области видимости подпрограммы, мы можем создать его внутри временного блока:

```
my @skipper_with_name;
{
    my @skipper = qw(голубая_рубашка шляпа накидка солнечные_очки крем);
    @skipper_with_name = ('Шкипер', \@skipper);
}
```

По выходе из блока второй элемент массива `@skipper_with_name` – это ссылка на массив, который внутри блока был известен под именем `@skipper`. Но теперь это имя лишено всякого смысла.

Однако все равно приходится вводить с клавиатуры слишком много ненужной информации только для того, чтобы сказать: «Второй элемент – это ссылка на массив, содержащий сведения о предметах экипировки». Мы можем создать ссылку на массив напрямую, для этого нужно лишь воспользоваться *конструктором анонимного массива*, который записывается в виде квадратных скобок:

```
my $ref_to_skipper_provisions =
    [ qw(голубая_рубашка шляпа накидка солнечные_очки крем) ];
```

Квадратные скобки принимают значения, заключенные в них (в списочном контексте), создают новый анонимный массив, инициализируют его заданными значениями и (что самое важное) возвращают ссылку на массив. То есть, как если бы мы написали:

```
my $ref_to_skipper_provisions;
{
    my @temporary_name =
        ( qw(голубая_рубашка шляпа накидка солнечные_очки крем) );
    $ref_to_skipper_provisions = \@temporary_name;
}
```

Здесь нам не нужно задумываться над временным именем массива и, кроме того, отпала необходимость во временном блоке. Результат работы конструктора анонимного массива является ссылкой на массив, которую можно использовать там же, где и скалярные величины.

Теперь, опираясь на это обстоятельство, мы можем конструировать большие списки:

```
my $ref_to_skipper_provisions =
    [ qw(голубая_рубашка шляпа накидка солнечные_очки крем) ];
my @skipper_with_name = ('Шкипер', $ref_to_skipper_provisions);
```

Разумеется, во временной скалярной переменной нет никакой необходимости. Мы можем вставить скаляр прямо в массив:

```
my @skipper_with_name = (
    'Шкипер',
    [ qw(голубая_рубашка шляпа накидка солнечные_очки крем) ];
);
```

Теперь посмотрим, что у нас получилось. Мы объявили массив `@skipper_with_name`, первый элемент которого – это имя Шкипера, а второй – ссылка на массив из пяти элементов. Таким образом, мы получили массив `@skipper_with_name`, содержащий два элемента, как и ранее.

Не следует путать квадратные скобки с круглыми. В данном примере они имеют разные предназначения. Если мы заменим квадратные скобки круглыми, то в результате получим список из шести элементов. Если заменим круглые скобки квадратными, то получим ссылку на анонимный массив с двумя элементами, которая будет записана в первый элемент массива `@skipper_with_name`.[1] Проще говоря, программный код:

```
my $fruits;
{
    my @secret_variable = ('ананас', 'папайя', 'манго');
    $fruits = \@secret_variable;
}
```

можно заменить на:

```
my $fruits = ['ананас', 'папайя', 'манго'];
```

Годится ли этот прием для создания более сложных структур? Конечно! Всякий раз, когда возникает необходимость в получении ссылки на массив, мы можем обратиться к конструктору анонимного массива. Фактически мы можем встроить их даже в наш пример со списками предметов экипировки:

```
sub get_provisions_list {
    return (
        ['Шкипер', [qw(голубая_рубашка шляпа накидка
                       солнечные_очки крем)]],
        ['Профессор', [qw(солнечные_очки фляжка_с_водой рулетка
                          батарейки радиоприемник)]],
        ['Джиллиган', [qw(красная_рубашка шляпа счастливые_носки
                          фляжка_с_водой)]],
          );
}

my @all_with_names = get_provisions_list( );
```

Если посмотреть на это извне, то мы получаем возвращаемое значение, состоящее из трех элементов. Каждый из этих элементов – это ссылка на массив, содержащий два элемента. Первый элемент каждого из массивов – это строка с именем персонажа, а второй – ссылка на анонимный массив переменной длины, содержащий список предметов экипировки. Всего этого нам удалось добиться, не придумывая имена промежуточных переменных и массивов.

---

1 В аудитории мы на собственном опыте не раз убеждались, что подобные ошибки чуть ли не самые распространенные при работе со ссылками.

С точки зрения вызывающей программы возвращаемое значение этой подпрограммы совершенно идентично ее предыдущей версии. А с точки зрения сопровождения программного кода нам удалось уменьшить визуальную помеху восприятия, ликвидировав ненужные промежуточные имена.

Существует возможность создавать ссылки на пустые анонимные массивы. Например, если нам потребуется добавить в список пассажиров «миссис Хоуэлл», которая решила отправиться на экскурсию налегке, мы можем просто написать:

```
['миссис Хоуэлл',
    [ ]
],
```

Это один из элементов длинного списка. Данный элемент представляет собой ссылку на массив с двумя элементами, первый из которых – это строка с именем, а второй – ссылка на пустой анонимный массив. Массив пуст просто потому, что миссис Хоуэлл решила отправиться в поездку налегке.

Создание анонимных хешей

Процедура создания анонимных хешей похожа на создание анонимных массивов. Рассмотрим список членов экипажа, который приводился в главе 4:

```
my %gilligan_info = (
    name => 'Джиллиган',
    hat => 'Белый',
    shirt => 'Красный',
    position => 'Первый помощник капитана',
);

my %skipper_info = (
    name => 'Шкипер',
    hat => 'Черный',
    shirt => 'Голубой',
    position => 'Капитан',
);

my @crew = (\%gilligan_info, \%skipper_info);
```

Переменные `%gilligan_info` и `%skipper_info` являются временными, они служат лишь для того, чтобы создать хеши, которые позднее войдут в состав конечной структуры данных. Создать ссылки на хеши можно посредством *конструктора анонимного хеша*, который записывается в форме фигурных скобок. Так, предыдущие строки мы можем заменить на:

```
my $ref_to_gilligan_info;
{
```

```
        my %gilligan_info = (
            name => 'Джиллиган',
            hat => 'Белый',
            shirt => 'Красный',
            position => 'Первый помощник капитана',
        );
        $ref_to_gilligan_info = \%gilligan_info;
    }
```

и, воспользовавшись конструктором анонимного хеша, на:

```
my $ref_to_gilligan_info = {
    name => 'Джиллиган',
    hat => 'Белый',
    shirt => 'Красный',
    position => 'Первый помощник капитана',
};
```

Значение, заключенное в фигурные скобки, – это список, состоящий из восьми элементов. Этот список преобразуется в анонимный хеш, содержащий четыре элемента (четыре пары ключ–значение). Perl берет ссылку на этот хеш и возвращает ее в виде скалярного значения, которое мы записываем в скалярную переменную. Таким образом, мы можем переписать программный код создания списка:

```
my $ref_to_gilligan_info = {
    name => 'Джиллиган',
    hat => 'Белый',
    shirt => 'Красный',
    position => 'Первый помощник капитана',
};

my $ref_skipper_info = {
    name => 'Шкипер',
    hat => 'Черный',
    shirt => 'Голубой',
    position => 'Капитан',
};

my @crew = ($ref_to_gilligan_info, $ref_to_skipper_info);
```

Как и ранее, мы можем вообще отказаться от переменных и сразу вставлять необходимые значения в список верхнего уровня:

```
my @crew = (
    {
        name => 'Джиллиган',
        hat => 'Белый',
        shirt => 'Красный',
        position => 'Первый помощник капитана',
    },
    {
        name => 'Шкипер',
        hat => 'Черный',
```

```
            shirt => 'Голубой',
            position => 'Капитан',
        },
    );
```

Обратите внимание: мы оставили запятые в списках после тех элементов, за которыми не следует сразу же закрывающая скобка. Этот стиль оформления хорош, поскольку облегчает сопровождение программного кода. Он позволяет добавлять новые элементы, переупорядочивать их или вставлять строки с комментариями, не нарушая целостность самих списков.

Теперь содержимое массива `@crew` совершенно идентично тому, что было получено ранее, но на этот раз нам не потребовалось придумывать имена промежуточных структур данных. Как и прежде, массив `@crew` состоит из двух элементов, каждый из которых представляет собой ссылку на хеш с информацией о конкретном члене экипажа.

Конструктор анонимных хешей всегда представляет свое содержимое в списочном контексте, а затем конструирует хеш в виде пар ключ–значение точно так же, как если бы мы создавали именованный хеш. Perl возвращает ссылку на этот хеш в виде единственного значения, которое может фигурировать там же, где и скалярные величины.

Теперь попробуем взглянуть на все это с точки зрения синтаксического анализатора: поскольку фигурные скобки служат как для оформления вложенных блоков программного кода, так и для конструирования анонимных хешей, компилятор может потребовать добавить дополнительные определения, которые будут свидетельствовать о ваших намерениях. Если компилятор выберет неправильное решение, вы можете помочь ему и явно сообщить, что предполагается получить. Чтобы подсказать компилятору, что некоторый синтаксический элемент является конструктором анонимного хеша, вставьте символ «плюс» (+) перед открывающей фигурной скобкой: `+{...}`. Чтобы подсказать компилятору, что он имеет дело с вложенным блоком программного кода, вставьте точку с запятой (которая представляет пустой оператор) в начало блока: `{;...}`.

Автовивификация

Теперь посмотрим еще раз на список предметов экипировки. Допустим, что содержимое этого списка должно извлекаться из такого файла:

```
Шкипер
    голубая_рубашка
    шляпа
    накидка
    солнечные_очки
    крем
Профессор
    солнечные_очки
```

```
        фляжка_с_водой
        рулетка
    Джиллиган
        красная рубашка
        шляпа
        счастливые_носки
        фляжка_с_водой
```

Перед названием каждого предмета присутствует отступ из нескольких пробелов. Строки, которые начинаются не с пробелов, соответствуют имени человека. Попробуем создать хеш предметов экипировки. Ключами хеша будут имена персонажей, а значениями – ссылки на массивы, содержащие списки предметов экипировки.

Информация из файла извлекается с помощью простого цикла:

```
my %provisions;
my $person;

while (<>) {
    if (/^(\S.*)/) { # имя персонажа (строка начинается не с пробелов)
        $person = $1;
        $provisions{$person} = [ ] unless exists $provisions{$person};
    } elsif (/^\s+(\S.*)/) { # предмет экипировки
        die 'Имя персонажа не определено!' unless defined $person;
        push @{ $provisions{$person} }, $1;
    } else {
        die "Не совсем понятно, что это такое: $_";
    }
}
```

В первую очередь мы объявили переменные, в которых будут храниться хеш с предметами экипировки и имя персонажа. После чтения очередной строки из файла выполняется проверка на наличие ведущих пробелов. Если была прочитана строка с именем персонажа, мы запоминаем ее и создаем элемент хеша для этого персонажа. Проверка `unless exists` позволяет избежать случайного удаления списка с предметами экипировки в том случае, если список предметов, принадлежащих одному и тому же персонажу, окажется разбитым на две части в исходном файле.

Например, впоследствии в конец файла могут быть добавлены две строки: «Шкипер» и «   секстан» (обратите внимание на наличие ведущих пробелов), которые только расширяют уже существующий список предметов, принадлежащих Шкиперу.

Ключом хеша является имя персонажа, а значением – ссылка на анонимный массив (изначально пустой). Если будет прочитана строка с названием предмета экипировки, она вставляется в конец нужного массива по ссылке на него.

Этот программный код прекрасно справляется с возложенной на него задачей, но он несколько избыточен. Почему? Потому что мы можем

опустить строку, где в элемент хеша записывается ссылка на пустой анонимный массив:

```
my %provisions;
my $person;

while (<>) {
    if (/^(\S.*)/) { # имя персонажа (строка начинается не с пробелов)
        $person = $1;
        ##$provisions{$person} = [ ] unless exists $provisions{$person};
    } elsif (/^\s+(\S.*)/) { # предмет экипировки
        die 'Имя персонажа не определено!' unless defined $person;
        push @{ $provisions{$person} }, $1;
    } else {
        die "Не совсем понятно, что это такое: $_";
    }
}
```

Как вы думаете, что произойдет, когда мы попытаемся добавить голубую рубашку в список предметов, принадлежащих Шкиперу? Если развернуть строку, которая вставляет название предмета в список, мы получим:

```
push @{ $provisions{'Шкипер'} }, "голубая_рубашка";
```

В этот момент элемент `$provisions{'Шкипер'}` еще не существует, однако мы пытаемся использовать его как ссылку на массив. Чтобы как-то разрешить сложившуюся ситуацию, Perl автоматически создаст ссылку на пустой анонимный массив, запишет ее в переменную и продолжит исполнение. После этого ссылка на только что созданный массив будет разыменована и в него будет добавлена голубая рубашка.

Этот процесс называется *автовивификацией* (*autovivification*). При попытке разыменовать отсутствующую переменную или переменную со значением `undef` для получения ссылки (формально это называется *контекстом l-значения* (*lvalue context*)) автоматически подставляется соответствующая ссылка на пустой элемент, что позволяет интерпретатору Perl продолжить работу.

Фактически мы всегда опирались на такой образ действия Perl, даже не подозревая об этом. Perl автоматически создает переменные, когда в этом возникает необходимость. Прежде элемент `$provisions{'Шкипер'}` не существовал, поэтому Perl создал его.

Следующий пример будет работать так, как ожидается:

```
my $not_yet; # новая неопределенная переменная
@$not_yet = (1, 2, 3);
```

Здесь мы попытались разыменовать значение `$not_yet`, как если бы это была ссылка на массив. Но поскольку изначально эта переменная имела значение `undef`, Perl выполнил действия, эквивалентные следующим строкам:

```
my $not_yet;
$not_yet = [ ]; # этот массив добавляется в результате
автовивификации
@$not_yet = (1, 2, 3);
```

Другими словами, изначально пустой массив превращается в массив из трех элементов.

Механизм автовивификации справляется даже с вложенными определениями:

```
my $top;
$top->[2]->[4] = 'lee-lou';
```

Изначально переменная `$top` содержит значение `undef`, но поскольку мы пытаемся разыменовать это значение как ссылку на массив, Perl автоматически создает и вставляет ссылку на пустой анонимный массив в переменную `$top`. После этого производится попытка обращения к третьему элементу массива (индекс 2), что вынуждает Perl увеличить размер массива до трех элементов. Данный элемент так же содержит значение `undef`, поэтому Perl создает и вставляет в него ссылку на еще один пустой анонимный массив. После этого второй массив увеличивается в размерах и в его пятый элемент записывается строка `'lee-lou'`.

Автовивификация и хеши

Аналогичным образом механизм автовивификации работает и в случае хешей. Если мы попытаемся разыменовать переменную, содержащую значение `undef`, как если бы это была ссылка на хеш, автоматически будет создан пустой анонимный хеш, а ссылка на него будет записана в эту переменную.

Механизм автовивификации обычно широко применяется в задачах предварительной обработки данных. Например, представьте себе, что Профессор протянул и запустил в эксплуатацию компьютерную сеть острова, и теперь у него появилось желание вести учет объема трафика между компьютерами. Он стал вносить в системный журнал записи, каждая из которых содержит имена узла-отправителя и узла-получателя и количество переданных байтов:

```
professor.hut gilligan.crew.hut 1250
professor.hut lovey.howell.hut 910
thurston.howell.hut lovey.howell.hut 1250
professor.hut lovey.howell.hut 450
professor.hut laser3.copyroom.hut 2924
ginger.girl.hut professor.hut 1218
ginger.girl.hut maryann.girl.hut 199
...
```

Теперь Профессор хочет получить сводную таблицу, которая содержала бы в каждой строке имя узла-отправителя, имя узла-получателя

и общее количество байтов, переданное за сутки. Создать такую таблицу нетрудно:

```
my %total_bytes;
while (<>) {
    my ($source, $destination, $bytes) = split;
    $total_bytes{$source}{$destination} += $bytes;
}
```

Теперь посмотрим, как работает этот код, когда получает первую строку данных. Итак, исполняется следующая строка:

```
$total_bytes{'professor.hut'}{'gilligan.crew.hut'} += 1250;
```

Поскольку изначально хеш `%total_bytes` пуст, Perl не найдет ключ `professor.hut` и поэтому создаст новый элемент со значением `undef` и вслед за этим сразу же попытается разыменовать его как ссылку на хеш. (Не забывайте,что в данном случае между двумя наборами фигурных скобок подразумевается оператор стрелки.) Perl запишет в этот элемент ссылку на новый пустой анонимный хеш, в который тут же будет добавлен элемент с ключом `gilligan.crew.hut`. Начальное значение этого элемента – `undef`, которое при попытке сложить его с числом 1250 интерпретируется как ноль; в результате в элемент хеша будет записано число 1250.

При анализе всех последующих строк журнала, которые содержат те же имена узла-отправителя и узла-получателя, будет производиться добавление количества байтов в один и тот же элемент хеша. Но всякий раз, когда будет встречаться новое имя узла-получателя, в хеш будет добавляться новый элемент со значением `undef` в счетчике байтов, а когда встретится новое имя узла-отправителя, с помощью механизма автовивификации будет создан хеш узла-получателя. Другими словами, все необходимое Perl сделает сам.

После того как из файла будут извлечены все данные, можно вывести сводную таблицу. В первой колонке таблицы будут выводиться имена узлов-отправителей:

```
for my $source (keys %total_bytes) {
...
```

Во второй колонке – имена узлов-получателей. Здесь есть небольшая хитрость. Нам нужны все ключи хеша, полученного в результате разыменования значения элемента первого хеша:

```
for my $source (keys %total_bytes) {
    for my $destination (keys %{ $total_bytes{$source} }) {
        ...
```

Вероятно, оба списка следовало бы отсортировать так, чтобы они оставались непротиворечивыми:

```
for my $source (sort keys %total_bytes) {
```

```
        for my $destination (sort keys %{ $total_bytes{$source} }) {
            print "$source => $destination:",
                  " $total_bytes{$source}{$destination} bytes\n";
        }
        print "\n";
    }
```

Это типичный подход к решению задачи вывода отчета.[1] Достаточно создать хеш хешей (возможно, с гораздо большей глубиной вложенности, пример такой структуры встретится нам позже), заполнить все уровни с помощью механизма автовивификации и затем отобразить полученные результаты.

Упражнения

Ответы на эти вопросы вы найдете в приложении A в разделе «Ответы к главе 5».

Упражнение 1 [5 мин]

Сможете ли вы найти ошибку в следующем отрывке программы, не запуская его? Если ошибку не удастся обнаружить за одну-две минуты, попробуйте отыскать ее, запустив отрывок.

```
my %passenger_1 = {
    name => 'Ginger',
    age => 22,
    occupation => 'Movie Star',
    real_age => 35,
    hat => undef,
};
my %passenger_2 = {
    name => 'Mary Ann',
    age => 19,
    hat => 'bonnet',
    favorite_food => 'corn',
};

my @passengers = (\%passenger_1, \%passenger_2);
```

Упражнение 2 [40 мин]

Файл Профессора с данными о трафике (*coconet.dat*, который упоминался выше) можно загрузить с веб-сайта O'Reilly. В этом файле имеются комментарии, начинающиеся с символа решетки (#). Перепишите пример анализа файла таким образом, чтобы программа корректно

---

1 Массу примеров структур данных можно найти в «Perl Data Structures Cookbook» и на страницах справочного руководства `perldsc`.

распознавала строки комментариев и пропускала их. (Подсказку вы найдете в самом комментарии, если прочитаете его!)

Видоизмените пример таким образом, чтобы при выводе группы списка с одним и тем же именем узла-отправителя в заголовке группы отображалось общее количество байтов, отправленных этим узлом. Отсортируйте список по числу отправленных байтов в порядке убывания. Каждую отдельную группу списка с узлами-получателями отсортируйте по числу принятых байтов от заданного узла-отправителя в порядке убывания.

В результате должен получиться список, в котором на первом месте должен стоять узел сети, отправивший больше всего байтов, а первым узлом-получателем в группе должен быть узел, получивший наибольшее количество байтов от данного узла-отправителя. После этого Профессор сможет воспользоваться полученными результатами для перестройки сети с целью повышения ее эффективности.

6

Управление сложными структурами данных

Итак, вы получили основные сведения о ссылках и можете перейти к рассмотрению дополнительных способов управления сложными структурами данных. Начнем с применения отладчика для изучения данных со сложной структурой, а затем покажем, как для тех же целей можно использовать модуль `Data::Dumper`. После этого вы научитесь легко и быстро сохранять и отыскивать данные со сложной структурой с помощью модуля `Storable`, а в конце главы мы дадим обзор возможностей `grep` и `map` при работе со сложными данными.

Использование отладчика для просмотра данных со сложной структурой

Отладчик Perl позволяет легко отобразить сложные данные. Например, попробуем выполнить программу подсчета трафика из главы 5 в пошаговом режиме:

```
my %total_bytes;
while (<>) {
    my ($source, $destination, $bytes) = split;
    $total_bytes{$source}{$destination} += $bytes;
}
for my $source (sort keys %total_bytes) {
    for my $destination (sort keys %{ $total_bytes{$source} }) {
        print "$source => $destination:",
              " $total_bytes{$source}{$destination} bytes\n";
    }
    print "\n";
}
```

В тестировании будут участвовать следующие данные:

```
professor.hut gilligan.crew.hut 1250
professor.hut lovey.howell.hut 910
thurston.howell.hut lovey.howell.hut 1250
professor.hut lovey.howell.hut 450
ginger.girl.hut professor.hut 1218
ginger.girl.hut maryann.girl.hut 199
```

Запустить программу в режиме отладки можно разными способами. Один из самых простых – вызвать Perl с ключом -d:

```
myhost% perl -d bytecounts bytecounts-in

Loading DB routines from perl5db.pl version 1.19
Editor support available.

Enter h or 'h h' for help, or 'man perldebug' for more help.

main::(bytecounts:2):        my %total_bytes;
  DB<1> s
main::(bytecounts:3):        while (<>) {
  DB<1> s
main::(bytecounts:4):          my ($source, $destination, $bytes) = split;
  DB<1> s
main::(bytecounts:5):         $total_bytes{$source}{$destination} += $bytes;
  DB<1> x $source, $destination, $bytes
0 'professor.hut'
1 'gilligan.crew.hut'
2 1250
```

Если вы собираетесь поэкспериментировать с программой самостоятельно, то должны знать, что каждая новая версия отладчика отличается от предыдущей, поэтому у себя на экране вы можете увидеть несколько иную картину. И еще: если у вас возникнут какие-либо затруднения, в любой момент вы можете ввести символ h, чтобы получить справку или заглянуть в perldoc perldebug.

Каждый раз, прежде чем исполнить очередную строку программного кода, отладчик выводит ее на экран. Это означает, что перед обращением к механизму автовивификации мы сможем увидеть состояние ключей хеша. Команда s выполняет один шаг, а команда x выводит список значений в удобочитаемом формате. С ее помощью мы можем убедиться, что переменные $source, $destination и $bytes содержат корректные значения. А теперь выполним второй проход цикла:

```
DB<2> s
  main::(bytecounts:3):      while (<>) {
```

Мы только что создали новый элемент хеша с помощью механизма автовивификации. Теперь посмотрим, что у нас получилось:

```
DB<2> x \%total_bytes
   0  HASH(0x132dc)
```

```
    'professor.hut' => HASH(0x37a34)
       'gilligan.crew.hut' => 1250
```

Если команде x передается ссылка на хеш, она выведет содержимое всего хеша в виде пар ключ–значение. Если какой-либо из элементов хеша является ссылкой, содержимое вложенного хеша также будет выведено на экран. В данном случае мы видим, что хеш %total_bytes имеет единственный ключ professor.hut, соответствующее значение которого является ссылкой на другой хеш. Вложенный хеш содержит единственный ключ gilligan.crew.hut со значением 1250.

Посмотрим, что произойдет после следующей операции присваивания:

```
DB<3> s
  main::(bytecounts:4):        my ($source, $destination, $bytes) = split;
DB<3> s
  main::(bytecounts:5):       $total_bytes{$source}{$destination} += $bytes;
DB<3> x $source, $destination, $bytes
  0 'professor.hut'
  1 'lovey.howell.hut'
  2 910
DB<4> s
  main::(bytecounts:3):      while (<>) {
DB<4> x \%total_bytes
  0  HASH(0x132dc)
 'professor.hut' => HASH(0x37a34)
    'gilligan.crew.hut' => 1250
    'lovey.howell.hut' => 910
```

Теперь мы добавили счетчик байтов, переданных от professor.hut к lovey.howell.hut. Хеш верхнего уровня при этом не изменился, но в хеше второго уровня появилась новая запись. Продолжим:

```
DB<5> s
  main::(bytecounts:4):        my ($source, $destination, $bytes) = split;
DB<6> s
  main::(bytecounts:5):       $total_bytes{$source}{$destination} += $bytes;
DB<6> x $source, $destination, $bytes
  0 'thurston.howell.hut'
  1 'lovey.howell.hut'
  2 1250
DB<7> s
  main::(bytecounts:3):      while (<>) {
DB<7> x \%total_bytes
  0 HASH(0x132dc)
   'professor.hut' => HASH(0x37a34)
      'gilligan.crew.hut' => 1250
      'lovey.howell.hut' => 910
   'thurston.howell.hut' => HASH(0x2f9538)
      'lovey.howell.hut' => 1250
```

Теперь стало гораздо интереснее! В хеше верхнего уровня появилась новая запись с ключом thurston.howell.hut и ссылка на новый хеш, соз-

данная механизмом автовивификации. Сразу же после создания в хеш была добавлена пара ключ–значение, свидетельствующая о том, что от `thurston.howell.hut` к `lovey.howell.hut` было передано 1250 байт. Сделаем еще один шаг:

```
DB<8> s
  main::(bytecounts:4):        my ($source, $destination, $bytes) = split;
DB<8> s
  main::(bytecounts:5):       $total_bytes{$source}{$destination} += $bytes;
DB<8> x $source, $destination, $bytes
  0 'professor.hut'
  1 'lovey.howell.hut'
  2 450
DB<9> s
  main::(bytecounts:3):      while (<>) {
DB<9> x \%total_bytes
  0 HASH(0x132dc)
   'professor.hut' => HASH(0x37a34)
      'gilligan.crew.hut' => 1250
      'lovey.howell.hut' => 1360
   'thurston.howell.hut' => HASH(0x2f9538)
      'lovey.howell.hut' => 1250
```

Теперь увеличился счетчик байтов, переданных от `professor.hut` к `lovey.howell.hut`. В общем ничего особенного. Сделаем еще шаг:

```
DB<10> s
  main::(bytecounts:4):        my ($source, $destination, $bytes) = split;
DB<10> s
  main::(bytecounts:5):       $total_bytes{$source}{$destination} += $bytes;
DB<10> x $source, $destination, $bytes
  0 'ginger.girl.hut'
  1 'professor.hut'
  2 1218
DB<11> s
  main::(bytecounts:3):      while (<>) {
DB<11> x \%total_bytes
  0 HASH(0x132dc)
   'ginger.girl.hut' => HASH(0x297474)
      'professor.hut' => 1218
   'professor.hut' => HASH(0x37a34)
      'gilligan.crew.hut' => 1250
      'lovey.howell.hut' => 1360
   'thurston.howell.hut' => HASH(0x2f9538)
      'lovey.howell.hut' => 1250
```

На этот раз добавилось новое имя узла-отправителя `ginger.girl.hut`. Обратите внимание: теперь хеш верхнего уровня содержит три элемента и все три элемента ссылаются на разные вложенные хеши. Сделаем еще шаг:

```
DB<12> s
  main::(bytecounts:4):        my ($source, $destination, $bytes) = split;
```

```
DB<12> s
  main::(bytecounts:5):       $total_bytes{$source}{$destination} += $bytes;
DB<12> x $source, $destination, $bytes
  0 'ginger.girl.hut'
  1 'maryann.girl.hut'
  2 199
DB<13> s
  main::(bytecounts:3):      while (<>) {
DB<13> x \%total_bytes
  0 HASH(0x132dc)
   'ginger.girl.hut' => HASH(0x297474)
      'maryann.girl.hut' => 199
      'professor.hut' => 1218
   'professor.hut' => HASH(0x37a34)
      'gilligan.crew.hut' => 1250
      'lovey.howell.hut' => 1360
   'thurston.howell.hut' => HASH(0x2f9538)
      'lovey.howell.hut' => 1250
```

Теперь в хеш был добавлен второй узел-получатель с информацией о количестве байтов, отправленных узлом `ginger.girl.hut`. Поскольку это была последняя строка в файле с данными (в данном примере), следующий шаг приводит к выходу из цикла `while`:

```
DB<14> s
  main::(bytecounts:8):      for my $source (sort keys %total_bytes) {
```

Мы не можем напрямую просмотреть элементы списка внутри круглых скобок, но можем отобразить его:

```
DB<14> x sort keys %total_bytes
    0 'ginger.girl.hut'
    1 'professor.hut'
    2 'thurston.howell.hut'
```

Этот список будем просматривать в цикле `for`. Все эти элементы являются именами узлов-отправителей, присутствующих в файле журнала. Ниже приводятся результаты выполнения следующего шага:

```
DB<15> s
    main::(bytecounts:9):    for my $destination (sort keys %{
$total_bytes{$source}
}) {
```

Начиная с этого момента мы можем точно определить, какие значения будут получены из списка, стоящего в круглых скобках:

```
DB<15> x $source
  0 'ginger.girl.hut'
DB<16> x $total_bytes{$source}
  0 HASH(0x297474)
   'maryann.girl.hut' => 199
   'professor.hut' => 1218
```

```
DB<18> x keys %{ $total_bytes{$source } }
  0 'maryann.girl.hut'
  1 'professor.hut'
DB<19> x sort keys %{ $total_bytes{$source } }
  0 'maryann.girl.hut'
  1 'professor.hut'
```

Обратите внимание: при выводе значения `$total_bytes{$source}` отладчик показал, что это ссылка на хеш. Кроме того, похоже, что добавление оператора `sort` никак не повлияло на порядок следования отображаемых данных, однако вы должны понимать, что вывод `keys` не всегда будет происходить в порядке сортировки. На следующем шаге производится вывод данных:

```
DB<20> s
  main::(bytecounts:10):     print "$source => $destination:",
  main::(bytecounts:11):       " $total_bytes{$source}{$destination} bytes\n"
;
DB<20> x $source, $destination
  0 'ginger.girl.hut'
  1 'maryann.girl.hut'
DB<21> x $total_bytes{$source}{$destination}
  0 199
```

Как видите, отладчик позволяет без особого труда просматривать даже структурированные данные, что существенно облегчает понимание принципа действия программы.

Просмотр данных со сложной структурой с помощью модуля Data::Dumper

Еще один способ визуализации данных со сложной структурой предоставляет базовый модуль `Data::Dumper` из дистрибутива Perl. Попробуем заменить заключительную часть программы подсчета трафика простым обращением к `Data::Dumper`:

```
use Data::Dumper;

my %total_bytes;
while (<>) {
    my ($source, $destination, $bytes) = split;
    $total_bytes{$source}{$destination} += $bytes;
}

print Dumper(\%total_bytes);
```

В модуле `Data::Dumper` определена подпрограмма с именем `Dumper`. Данная подпрограмма напоминает по своему действию команду `x` отладчика. Мы можем передать подпрограмме `Dumper` одно или более значений, а она в свою очередь преобразует эти значения в строку, готовую к выводу на экран. Разница между командой `x` отладчика и подпрограммой

Dumper состоит в том, что строка, создаваемая подпрограммой, является программным кодом на языке Perl:

```
myhost% perl bytecounts2 <bytecounts-in
$VAR1 = {
    'thurston.howell.hut' => {
                              'lovey.howell.hut' => 1250
                             },
    'ginger.girl.hut' => {
                          'maryann.girl.hut' => 199,
                          'professor.hut' => 1218
                         },
    'professor.hut' => {
                        'gilligan.crew.hut' => 1250,
                        'lovey.howell.hut' => 1360
                       }
        };
myhost%
```

Этот программный код достаточно легко читается. Он показывает, что у нас имеется ссылка на хеш, состоящий из трех элементов, значением каждого из которых является ссылка на вложенный хеш. Мы можем исполнить этот код и получить хеш, эквивалентный оригиналу. Однако пока не стоит рассматривать эту возможность как способ сохранения данных со сложной структурой между вызовами программы.

Модуль Data::Dumper, как и команда x отладчика, прекрасно справляется с перекрестными ссылками. Чтобы продемонстрировать это, вернемся к примеру из главы 5, страдающему утечкой памяти:

```
use Data::Dumper;
$Data::Dumper::Purity = 1; # включить возможность работы
                           # с перекрестными ссылками
my @data1 = qw(one won);
my @data2 = qw(two too to);
push @data2, \@data1;
push @data1, \@data2;
print Dumper(\@data1, \@data2);
```

Ниже приводится результат работы этой программы:

```
$VAR1 = [
         'one',
         'won',
         [
          'two',
          'too',
          'to',
          [ ]
         ]
        ];
$VAR1->[2][3] = $VAR1;
$VAR2 = $VAR1->[2];
```

Обратите внимание, как подпрограмма Dumper создала переменные. Переменная $VAR1 соответствует ссылке на @data1, а $VAR2 — ссылке на @data2. Отладчик отобразит эти данные аналогичным образом:

```
DB<1> x \@data1, \@data2
    0 ARRAY(0xf914)
  0 'one'
  1 'won'
  2 ARRAY(0x3122a8)
    0 'two'
    1 'too'
    2 'to'
    3 ARRAY(0xf914)
      -> REUSED_ADDRESS
    1 ARRAY(0x3122a8)
-> REUSED_ADDRESS
```

В данном случае фраза REUSED_ADDRESS указывает на то, что некоторые части данных фактически являются ссылками на элементы, которые уже были показаны.

YAML

Однако модуль Data::Dumper — не единственное, с чем можно поиграть на острове. Брайан Ингерсон (Brian Ingerson) придумал язык разметки YAML (Yet Another Markup Language – еще один язык разметки), с помощью которого можно выводить данные в более удобочитаемом (и более компактном) виде. Данные на этом языке отображаются тем же способом, что и с помощью Data::Dumper. Позднее, когда будем вести более детальное обсуждение модулей, мы познакомимся с этим языком поближе, а пока не будем слишком углубляться.

Вернемся к предыдущему примеру и вместо Data::Dumper подставим YAML, а вместо вызова подпрограммы Dumper() будем вызывать подпрограмму Dump().

```
use YAML;

my %total_bytes;

while (<>) {
    my ($source, $destination, $bytes) = split;
    $total_bytes{$source}{$destination} += $bytes;
}

print Dump(\%total_bytes);
```

Если воспользоваться теми же самыми исходными данными, будет получен следующий результат:

```
--- #YAML:1.0
ginger.girl.hut:
  maryann.girl.hut: 199
  professor.hut: 1218
```

```
professor.hut:
  gilligan.crew.hut: 1250
  lovey.howell.hut: 1360
thurston.howell.hut:
  lovey.howell.hut: 1250
```

Теперь информация читается намного легче и занимает меньше места на экране, что действительно удобно при исследовании глубоко вложенных структур данных.

Сохранение данных со сложной структурой с помощью модуля Storable

Мы можем взять результат работы подпрограммы `Dumper` из модуля `Data::Dumper`, записать его в файл, а затем загрузить файл из другой программы. После того как программный код на языке Perl будет исполнен, мы получим две переменные, `$VAR1` и `$VAR2`, которые будут полностью эквивалентны первоначальным данным. Этот прием называется *маршаллингом*[1] (*marshaling*) данных: преобразование исходных данных в форму, в которой они могут быть записаны в файл в виде последовательности байтов и из которой позднее они могут быть восстановлены.

Однако для нужд маршаллинга удобнее другой базовый модуль Perl: `Storable`. В сравнении с `Data::Dumper` модуль `Storable` порождает меньшие по объему файлы. (Модуль `Storable` совсем недавно вошел в состав дистрибутива Perl, и его можно установить из CPAN.)

Интерфейс модуля `Storable` напоминает `Data::Dumper` за исключением того, что все входные аргументы передаются подпрограмме в виде одной ссылки. Попробуем сохранить в файл структуры данных, имеющие перекрестные ссылки:

```
use Storable;
my @data1 = qw(one won);
my @data2 = qw(two too to);
push @data2, \@data1;
push @data1, \@data2;
store [\@data1, \@data2], 'some_file';
```

В результате будет получен файл длиной менее 100 байт, что намного меньше, чем было получено с помощью `Data::Dumper`. Кроме того, файл имеет удобочитаемый формат. Прочитать файл можно достаточно просто средствами модуля `Storable`.[2] Выборка данных из файла также мо-

---

1 В других языках и технологиях программирования такие преобразования более известны под термином *сериализация* (*десериализация* – при восстановлении) *данных*. – *Примеч. науч. ред.*

2 Формат представления данных модулем `Storable` по умолчанию зависит от аппаратной архитектуры. В документации к модулю показано, как создавать файлы, не зависящие от порядка следования байтов.

жет выполняться с помощью этого же модуля. Результатом будет единственная ссылка на массив. Посмотрим на полученные результаты, чтобы убедиться, что данные были сохранены корректно:

```
use Storable;
my $result = retrieve 'some_file';
use Data::Dumper;
$Data::Dumper::Purity = 1;
print Dumper($result);
```

Ниже приводятся полученные результаты:

```
$VAR1 = [
          [
            'one',
            'won',
            [
              'two',
              'too',
              'to',
              [ ]
            ]
          ],
          [ ]
        ];
$VAR1->[0][2][3] = $VAR1->[0];
$VAR1->[1] = $VAR1->[0][2];
```

Функционально получилась структура данных, идентичная оригинальной. Здесь мы видим две ссылки на массивы внутри одного массива верхнего уровня. Чтобы получить нечто более похожее на то, что мы видели в самом начале, опишем ожидаемый результат более явно:

```
use Storable;
my ($arr1, $arr2) = @{ retrieve 'some_file' };
use Data::Dumper;
$Data::Dumper::Purity = 1;
print Dumper($arr1, $arr2);
```

или, что то же самое:

```
use Storable;
my $result = retrieve 'some_file';
use Data::Dumper;
$Data::Dumper::Purity = 1;
print Dumper(@$result);
```

в результате получим:

```
$VAR1 = [
          'one',
          'won',
          [
            'two',
```

```
            'too',
            'to',
            [ ]
          ]
        ];
$VAR1->[2][3] = $VAR1;
$VAR2 = $VAR1->[2];
```

практически то же самое, что и в первой программе. Посредством модуля Storable мы можем сохранять данные в файле, а затем извлекать их из файла. Более подробную информацию о модуле Storable вы найдете, как обычно, в документации perldoc Storable.

Операторы grep и map

По мере роста степени сложности структур данных все большее значение начинают приобретать конструкции высокого уровня, позволяющие решать задачи по извлечению и преобразованию данных. В этом отношении операторы языка Perl grep и map обладают непревзойденными возможностями.

Обходное решение

Некоторые проблемы, на первый взгляд достаточно сложные, перестают быть таковыми после того, как будет найдено одно или два решения. Допустим, что необходимо определить, существуют ли в списке числа, имеющие нечетную сумму цифр, причем сами числа нас не интересуют – нам лишь нужно знать, есть такие числа в списке или нет.

В таких случаях, как правило, достаточно найти косвенное решение.[1] В первую очередь перед нами встает проблема извлечения данных, поэтому мы воспользуемся оператором grep. Попробуем отбирать данные не по значениям, а по индексам:

```
my @input_numbers = (1, 2, 4, 8, 16, 32, 64);
my @indices_of_odd_digit_sums = grep {
    ...
} 0..$#input_numbers;
```

Здесь выражение 0..$#input_numbers представляет список индексов в массиве. Внутри блока переменная $_ может принимать целые значения от 0 до 6 (всего семь элементов). Теперь нас не интересует, имеет ли число $_ нечетную сумму цифр. Нас интересует вопрос, имеет ли элемент массива с индексом $_ нечетную сумму цифр. Поэтому вместо $_ мы будем рассматривать число $input_numbers[$_]:

---

1 Известный принцип вычислительной техники гласит: «Нет такой задачи, которую нельзя было бы решить введением дополнительных уровней косвенности». Разумеется, с ростом косвенности уменьшается очевидность решения, поэтому здесь важно не перестараться и найти золотую середину.

```
my @indices_of_odd_digit_sums = grep {
    my $number = $input_numbers[$_];
    my $sum;
    $sum += $_ for split //, $number;
    $sum % 2;
} 0..$#input_numbers;
```

В результате мы получим индексы чисел 1, 16 и 32: 0, 4 и 5. По значениям этих индексов можно извлекать из массива сами числа:

```
my @odd_digit_sums = @input_numbers[ @indices_of_odd_digit_sums ];
```

Принцип решения здесь заключается в том, чтобы использовать `$_` для хранения идентификатора конкретного элемента, как, например, ключ хеша или индекс массива, а затем с помощью этого идентификатора получить доступ к требуемым значениям внутри блока или выражения.

Ниже приводится еще один пример: из массива `@x` необходимо выбрать все элементы, значения которых больше, чем значения соответствующих им элементов в массиве `@y`. Здесь нам снова понадобится переменная `$_` для индексации массива `@x`:

```
my @bigger_indices = grep {
    if ($_ > $#y or $x[$_] > $y[$_]) {
        1; # да, выбрать
    } else {
        0; # нет, не выбирать
    }
} 0..$#x;
my @bigger = @x[@bigger_indices];
```

В блоке оператора `grep` переменная `$_` будет приобретать значения от 0 до наибольшего индекса в массиве `@x`. Если индекс элемента в массиве `@x` превысит размерность массива `@y`, мы автоматически выбираем его. В противном случае сравниваем соответствующие значения элементов двух массивов, выбирая индекс только в том случае, если результат сравнения соответствует заданному условию.

Этот пример несколько избыточен. Вместо значений 0 и 1 мы могли бы просто возвращать результат логического выражения:

```
my @bigger_indices = grep {
    $_ > $#y or $x[$_] > $y[$_];
} 0..$#x;
my @bigger = @x[@bigger_indices];
```

Можно поступить еще проще – опустить шаг создания промежуточного массива и сразу же возвращать отобранные значения элементов с помощью оператора `map`:

```
my @bigger = map {
    if ($_ > $#y or $x[$_] > $y[$_]) {
        $x[$_];
    } else {
```

```
          ( );
      }
  } 0..$#x;
```

Если элемент с заданным индексом подпадает под заданный критерий, возвращается значение этого элемента. В противном случае возвращается пустой список.

Выбор и модификация данных со сложной структурой

Мы можем применить операторы grep и map для работы с более сложными структурами данных. Возьмем для примера список предметов экипировки из главы 5:

```
my %provisions = (
    'Шкипер' => [qw(голубая_рубашка шляпа накидка солнечные_очки крем)],
    'Профессор' => [qw(солнечные_очки фляжка_с_водой рулетка
                      батарейки радиоприемник)],
    'Джиллиган' => [qw(красная_рубашка шляпа счастливые_носки
                      фляжка_с_водой)],
);
```

В этом случае выражение $provisions{"Профессор"} вернет ссылку на массив с предметами экипировки, принадлежащими Профессору, а выражение $provisions{"Джиллиган"}[-1] – последний предмет, который взял с собой Джиллиган.

Попробуем ответить на некоторые вопросы, касающиеся этих данных. Вопрос первый: кто взял с собой меньше пяти предметов?

```
my @packed_light = grep @{ $provisions{$_} } < 5, keys %provisions;
```

В этом случае $_ содержит имя персонажа. Мы берем это имя, отыскиваем ссылку на массив со списком предметов, разыменовываем ее в скалярном контексте, что дает нам количество предметов, и затем сравниваем его с числом 5. Неужели вы не знаете кто это – это же Джиллиган.

Следующий вопрос чуть сложнее. Кто взял с собой фляжку с водой?

```
my @all_wet = grep {
  my @items = @{ $provisions{$_} };
  grep $_ eq 'фляжка_с_водой', @items;
} keys %provisions;
```

Вновь начав со списка имен (keys %provisions), мы извлекаем все предметы экипировки, а затем, читая полученный список, с помощью внутреннего оператора grep подсчитываем количество элементов, содержащих строку 'фляжка_с_водой'. Если это число равно 0, значит, фляжка у данного персонажа отсутствует, то есть этот результат будет интерпретироваться внешним оператором grep как «ложь». Если число не равно нулю, значит, фляжка у текущего персонажа имеется, и резуль-

тат выражения будет интерпретироваться внешним оператором grep как «истина». Теперь мы видим, что Шкипера будет мучить жажда, если кто-нибудь не придет ему на помощь.

Кроме этого мы можем выполнить некоторые преобразования. Например, превратим этот хеш в список ссылок на массивы, каждый из которых будет состоять из двух элементов. Первый элемент – это имя персонажа, а второй – ссылка на массив с предметами экипировки, принадлежащими данному персонажу:

```
my @remapped_list = map {
    [ $_ => $provisions{$_} ];
} keys %provisions;
```

Ключами в хеше %provisions являются имена персонажей. Для каждого персонажа создается список из двух элементов с именем и со ссылкой на массив, содержащий предметы экипировки. Этот список находится внутри конструктора анонимного массива, в результате для каждого персонажа мы получаем ссылку на вновь созданный массив. Три персонажа на входе – три ссылки на выходе.[1] Теперь попробуем пойти другим путем. Превратим хеш в серию ссылок на массивы. Каждый массив будет содержать имя персонажа и один из предметов экипировки, принадлежащий ему:

```
my @person_item_pairs = map {
  my $person = $_;
  my @items = @{ $provisions{$person} };
  map [$person => $_], @items;
} keys %provisions;
```

Да-да! Один оператор map внутри другого! Внешний оператор map отбирает по одному персонажу за раз. Мы запоминаем его имя в переменной $person, а затем извлекаем из хеша список предметов. Внутренний оператор map обходит список предметов и создает для каждого элемента ссылку на анонимный массив. В результате все анонимные массивы будут содержать имя персонажа и один из предметов его экипировки.

Мы вынуждены были ввести промежуточную переменную $person для запоминания значения переменной $_ внешнего оператора map, поскольку мы не можем одновременно обращаться к временным переменным $_ внешнего и внутреннего операторов map.

Упражнения

Ответы на эти вопросы вы найдете в приложении A в разделе «Ответы к главе 6».

---

[1] Если бы мы опустили внутренние скобки, то получили бы на выходе список из шести элементов. Это имело бы смысл, если бы мы собирались создать из списка еще один хеш.

Упражнение 1 [20 мин]

Программа из упражнения 2 главы 5 при каждом запуске читает файл журнала целиком. Однако Профессор получает новый файл журнала каждый день и не желает хранить весь объем данных в одном большом файле, анализ которого с каждым днем будет занимать все больше и больше времени.

Перепишите программу таким образом, чтобы информацию о трафике, полученную ранее, она сохраняла в отдельном файле. Тогда Профессор сможет запускать программу ежедневно, передавая ей очередную порцию данных, и получать обновленные сведения о трафике.

Упражнение 2 [5 мин]

Какие еще полезные функциональные возможности можно было бы добавить в программу? Просто подумайте об этом, не надо их реализовывать!

7

Ссылки на подпрограммы

До сих пор мы рассматривали ссылки на три основных типа данных языка Perl – скаляры, массивы и хеши. Но есть возможность получать ссылки и на подпрограммы.

Для чего это нужно? Скажем так: ссылки на массивы позволяют выполнять однотипные действия над разными массивами в разное время, а ссылки на подпрограммы позволяют из одного и того же программного кода вызывать разные подпрограммы в разное время. Кроме того, ссылки позволяют конструировать очень сложные структуры данных. Ссылки на подпрограммы дают возможность сделать подпрограммы частью этих сложных структур данных.

Иначе говоря, переменные или структуры данных являются хранилищами информации в программе, а ссылки на подпрограммы аналогичным образом можно назвать хранилищами поведенческих реакций программы. Примеры из этого раздела помогут вам разобраться во всех этих хитросплетениях.

Ссылки на именованные подпрограммы

Между Шкипером и Джиллиганом состоялся следующий диалог:

```
sub skipper_greets {
    my $person = shift;
    print "Шкипер: Эй, $person!\n";
}

sub gilligan_greets {
    my $person = shift;
    if ($person eq "Шкипер") {
        print "Джиллиган: Сэр, $person!\n";
      } else {
        print "Джиллиган: Привет, $person!\n";
```

```
    }
}

skipper_greets("Джиллиган");
gilligan_greets("Шкипер");
```

В результате получилось следующее:

```
Шкипер: Эй, Джиллиган!
Джиллиган: Сэр, Шкипер!
```

Вроде бы ничего необычного, однако обратите внимание: Джиллиган ведет себя по-разному в зависимости от того, кто к нему обратился, Шкипер или кто-либо другой.

Теперь представьте, что в хижину входит Профессор и оба члена экипажа приветствуют его:

```
skipper_greets('Профессор');
gilligan_greets('Профессор');
```

В результате получится следующее:

```
Шкипер: Эй, Профессор!
Джиллиган: Привет, Профессор!
```

Профессор должен как-то ответить на приветствие:

```
sub professor_greets {
    my $person = shift;
    print "Профессор: Насколько я понимаю, вы $person!\n";
}

professor_greets('Джиллиган');
professor_greets('Шкипер');
```

В результате ответ прозвучит следующим образом:

```
Профессор: Насколько я понимаю, вы Джиллиган!
Профессор: Насколько я понимаю, вы Шкипер!
```

Гмм! Не очень удобно. Когда в хижину входит новый персонаж, поведение которого описывается отдельной именованной подпрограммой, нам придется точно указывать, какую подпрограмму следует вызвать. Конечно, это можно сделать, пусть и за счет большого объема программного кода, сложного для сопровождения, но мы можем существенно упростить задачу, лишь чуть-чуть добавив косвенности, как мы делали это при работе с массивами и хешами.

Прежде всего, воспользуемся оператором взятия ссылки. Мы не будем давать дополнительных пояснений, поскольку это тот же обратный слэш, с которым вы уже знакомы:

```
my $ref_to_greeter = \&skipper_greets;
```

Здесь мы взяли ссылку на подпрограмму `skipper_greets()`. Обратите внимание на предшествующий символ амперсанда – его наличие со-

вершенно необходимо, так же как и отсутствие круглых скобок. Значение ссылки сохраняется в переменной `$ref_to_greeter`, которая может использоваться везде, где допускается использование обычной скалярной величины.

Есть только одна причина, чтобы вернуться к оригинальной подпрограмме при разыменовании ссылки, – это вызвать ее. Разыменование ссылки на подпрограмму напоминает процедуру разыменования любых других ссылок. Для начала рассмотрим способ, которым мы воспользовались бы, ничего не зная о ссылках (включая необязательный символ амперсанда):

```
& skipper_greets ( 'Джиллиган' )
```

Теперь заменим имя подпрограммы на имя ссылки, окруженное фигурными скобками:

```
& { $ref_to_greeter } ( 'Джиллиган' )
```

Эта конструкция вызывает подпрограмму, на которую в настоящий момент ссылается `$ref_to_greeter`, и передает ей единственный аргумент: имя Джиллиган.

Однако такая форма записи выглядит несколько уродливо, вам не кажется? К счастью, здесь применимы те же правила упрощения, что и для обычных ссылок. Если в фигурных скобках стоит обычная скалярная переменная, их можно просто опустить:

```
& $ref_to_greeter ( 'Джиллиган' )
```

Можно даже воспользоваться оператором стрелки:

```
$ref_to_greeter -> ( 'Джиллиган' )
```

Последняя форма записи особенно удобна, когда ссылка находится внутри большой структуры данных, в чем вы вскоре сможете убедиться.

Чтобы Джиллиган и Шкипер смогли поприветствовать Профессора, нам достаточно вызвать все подпрограммы в цикле:

```
for my $greet (\&skipper_greets, \&gilligan_greets) {
    $greet->('Профессор');
}
```

В первую очередь здесь создается список из двух элементов, каждый из которых является ссылкой на подпрограмму. После этого каждая ссылка разыменовывается, в результате чего происходит вызов той или иной подпрограммы, которым в виде аргумента передается строка `Профессор`.

Мы уже видели ссылки на подпрограммы в виде скалярной переменной и в виде элемента списка. А возможно ли поместить ссылку на подпрограмму в сложную структуру данных? Разумеется. Попробуем

создать таблицу, в которой определим соответствие между именами персонажей и их реакцией на приветствие других людей, а затем перепишем предыдущий пример, основываясь на этой таблице:

```
sub skipper_greets {
    my $person = shift;
    print "Шкипер: Эй, $person!\n";
}

sub gilligan_greets {
    my $person = shift;
    if ($person eq 'Шкипер') {
        print "Джиллиган: Сэр, $person!\n";
      } else {
        print "Джиллиган: Привет, $person!\n";
    }
}

sub professor_greets {
    my $person = shift;
    print "Профессор: Насколько я понимаю, вы $person!\n";
}

my %greets = (
    Gilligan  => \&gilligan_greets,
    Skipper  => \&skipper_greets,
    Professor => \&professor_greets,
);

for my $person (qw(Skipper Gilligan)) {
    $greets{$person}->('Профессор');
}
```

Обратите внимание: в переменной `$person` находится имя персонажа, по которому производится поиск требуемой ссылки на подпрограмму в хеше. После того как ссылка будет получена, мы разыменовываем ее и передаем подпрограмме имя персонажа, которого хотим поприветствовать. В результате мы получаем корректное поведение:

```
Шкипер: Эй, Профессор!
Джиллиган: Привет, Профессор!
```

Теперь попробуем реализовать ситуацию, когда все присутствующие приветствуют друг друга:

```
sub skipper_greets {
    my $person = shift;
    print "Шкипер: Эй, $person!\n";
}

sub gilligan_greets {
    my $person = shift;
    if ($person eq 'Шкипер') {
        print "Джиллиган: Сэр, $person!\n";
```

```
      } else {
        print "Джиллиган: Привет, $person!\n";
    }
}

sub professor_greets {
    my $person = shift;
    print "Профессор: Насколько я понимаю, вы $person!\n";
}

my %greets = (
    Gilligan  => \&gilligan_greets,
    Skipper  => \&skipper_greets,
    Professor  => \&professor_greets,
);

my @everyone = sort keys %greets;
for my $greeter (@everyone) {
    for my $greeted (@everyone) {
        $greets{$greeter}->($greeted)
           unless $greeter eq $greeted; # не стоит приветствовать себя самого
    }
}
```

Результат работы этой программы выглядит следующим образом:

```
Джиллиган: Привет, Профессор!
Джиллиган: Сэр, Шкипер!
Профессор: Насколько я понимаю, вы Джиллиган!
Профессор: Насколько я понимаю, вы Шкипер!
Шкипер: Эй, Джиллиган!
Шкипер: Эй, Профессор!
```

Что-то слишком сложно. Давайте позволим им входить в хижину по одному:

```
sub skipper_greets {
    my $person = shift;
    print "Шкипер: Эй, $person!\n";
}

sub gilligan_greets {
    my $person = shift;
    if ($person eq 'Шкипер') {
        print "Джиллиган: Сэр, $person!\n";
      } else {
        print "Джиллиган: Привет, $person!\n";
    }
}

sub professor_greets {
    my $person = shift;
    print "Профессор: Насколько я понимаю, вы $person!\n";
}
```

```
my %greets = (
    «Джиллиган» => \&gilligan_greets,
    «Шкипер» => \&skipper_greets,
    «Профессор» => \&professor_greets,
);

my @room; # сначала в хижине никого нет
for my $person (qw(Джиллиган Шкипер Профессор)) {
    print "\n";
    print "В хижину входит $person.\n";
    for my $room_person (@room) {
        $greets{$person}->($room_person); # приветствует
        $greets{$room_person}->($person); # получает ответ
    }
    push @room, $person; # входит и устраивается поудобнее
}
```

Так начинается обычный день на тропическом острове:

```
В хижину входит Джиллиган.

В хижину входит Шкипер.
Шкипер: Эй, Джиллиган!
Джиллиган: Сэр, Шкипер!

В хижину входит Профессор.
Профессор: Насколько я понимаю, вы Джиллиган!
Джиллиган: Привет, Профессор!
Профессор: Насколько я понимаю, вы Шкипер!
Шкипер: Эй, Профессор!
```

Анонимные подпрограммы

В последнем примере мы нигде явно[1] не обращались к таким подпрограммам, как `professor_greets()` или `skipper_greets()`. Мы вызывали их косвенно – через ссылки на подпрограммы. Однако нам пришлось приложить определенные усилия, чтобы придумать имена подпрограммам и инициализировать структуру данных. Но мы могли бы создать анонимные подпрограммы точно так же, как анонимные массивы или хеши!

Добавим в компанию обитателей острова еще один персонаж, Джинджер, и определим ее поведение с помощью анонимной подпрограммы:

```
my $ginger = sub {
    my $person = shift;
    print "Джинджер: (томным голосом) О! Салют, $person!\n";
};
$ginger->('Шкипер');
```

[1] По имени. – *Примеч. науч. ред.*

Анонимная подпрограмма объявляется так же, как и обычная, но между ключевым словом `sub` и открывающей скобкой блока программного кода нет имени. Кроме того, такая форма записи расценивается как часть выражения, поэтому объявление анонимной подпрограммы должно завершаться точкой с запятой или иным разделителем выражений, как и в большинстве случаев.

```
sub { ... тело подпрограммы ... };
```

Значением переменной `$ginger` является ссылка на подпрограмму, как если бы мы объявили обычную подпрограмму, а потом взяли бы ссылку на нее. Дойдя до последнего выражения, мы увидим такое приветствие:

```
Джинджер: (томным голосом) О! Салют, Шкипер!
```

Сейчас мы сохранили ссылку на подпрограмму в скалярной переменной, но мы можем поместить объявления `sub { ... }` прямо в хеш:

```
my %greets = (

    'Шкипер' => sub {
        my $person = shift;
        print "Шкипер: Эй, $person!\n";
    },

    'Джиллиган' => sub {
        my $person = shift;
        if ($person eq 'Шкипер') {
            print "Джиллиган: Сэр, $person!\n";
          } else {
            print "Джиллиган: Привет, $person!\n";
        }
    },

    'Профессор' => sub {
        my $person = shift;
        print "Профессор: Насколько я понимаю, вы $person!\n";
    },

    'Джинджер' => sub {
        my $person = shift;
        print "Джинджер: (томным голосом) О! Салют, $person!\n";
    },
);

my @room; # сначала в хижине никого нет
for my $person (qw(Джиллиган Шкипер Профессор Джинджер)) {
    print "\n";
    print "В комнату входит $person.\n";
    for my $room_person (@room) {
        $greets{$person}->($room_person); # приветствует
        $greets{$room_person}->($person); # получает ответ
    }
    push @room, $person; # входит и устраивается поудобнее
}
```

Обратите внимание, насколько проще стал выглядеть программный код. Объявления подпрограмм находятся прямо в структуре данных. Результат работы программы вполне предсказуем:

```
В комнату входит Джиллиган.

В комнату входит Шкипер.
Шкипер: Эй, Джиллиган!
Джиллиган: Сэр, Шкипер!

В комнату входит Профессор.
Профессор: Насколько я понимаю, вы Джиллиган!
Джилиган: Привет, Профессор!
Профессор: Насколько я понимаю, вы Шкипер!
Шкипер: Эй, Профессор!

В комнату входит Джинджер.
Джинджер: (томным голосом) О! Здравствуйте, Джиллиган!
Джилиган: Привет, Джинджер!
Джинджер: (томным голосом) О! Здравствуйте, Шкипер!
Шкипер: Эй, Джинджер!
Джинджер: (томным голосом) О! Здравствуйте, Профессор!
Профессор: Насколько я понимаю, вы Джинджер!
```

Чтобы добавить еще один персонаж, достаточно вставить в хеш еще одно объявление подпрограммы, определяющей его поведение, и занести имя персонажа в список лиц, входящих в хижину. Такую масштабируемость мы получили потому, что определили поведенческие реакции персонажей как обычные данные, которые можно обойти и выбрать в цикле благодаря ссылкам на подпрограммы.

Подпрограммы обратного вызова

Очень часто ссылки на подпрограммы служат для организации *обратных вызовов*. Подпрограмма обратного вызова определяет, что необходимо сделать, когда программа достигает определенной точки в алгоритме.

Например, модуль `File::Find` экспортирует подпрограмму `find`, которая выполняет обход дерева каталогов способом, не зависящим от платформы. В простейшем случае подпрограмме `find` передаются два аргумента: начальный каталог и описание того, «что следует сделать» с именем каждого файла или каталога, которые будут обнаружены внутри заданного. В данном случае под «что сделать» подразумевается ссылка на подпрограмму:

```
use File::Find;
sub what_to_do {
    print "Найден файл или каталог $File::Find::name\n";
}
my @starting_directories = qw(.);

find(\&what_to_do, @starting_directories);
```

В этом примере поиск начинается от текущего каталога (.), в процессе которого отыскиваются все находящиеся внутри него файлы и подкаталоги. Для каждого найденного элемента вызывается подпрограмма what_to_do(), которой через глобальные переменные передаются несколько аргументов. В данном случае в переменной File::Find::name содержится полный путь к найденному файлу или каталогу (начиная от исходного каталога).

В этом примере мы передали подпрограмме find в качестве аргументов список каталогов поиска и реакцию на найденный элемент.

Не имеет большого смысла придумывать название для подпрограммы, которая вызывается лишь в одном месте, поэтому перепишем предыдущий пример, оставив подпрограмму анонимной:

```
use File::Find;
my @starting_directories = qw(.);

find(
    sub {
        print "Найден файл или каталог $File::Find::name\n";
    },
    @starting_directories,
);
```

Замыкания

Модуль File::Find позволяет получать дополнительные сведения о файлах, например их размеры. Для удобства разработки подпрограмм обратного вызова имя элемента, найденного в текущем рабочем каталоге, содержится в переменной $_.

Вы могли заметить, что в предыдущем примере мы извлекали имя элемента из переменной $File::Find::name. Как же определить, где находится настоящее имя элемента: в $_ или в $File::Find::name? В переменной $File::Find::name содержится полное имя найденного элемента относительно начального каталога поиска. Когда производится обращение к подпрограмме обратного вызова, рабочим каталогом считается тот, в котором обнаружен очередной элемент. Предположим, что требуется отыскать файлы в текущем рабочем каталоге, поэтому в качестве начального каталога мы задаем его имя ("."). Если в момент вызова подпрограммы find текущим был каталог /usr, она начнет поиск файлов в этом и всех вложенных подкаталогах этого каталога. Когда find найдет файл /usr/bin/perl, текущим рабочим каталогом (в момент обращения к подпрограмме обратного вызова) будет /usr/bin. В этом случае в переменной $_ будет храниться имя perl, а в переменной $File::Find::name — ./bin/perl, то есть путь к файлу относительно начального каталога поиска.

Все это означает, что проверки, выполняемые над файлом, такие как -s, автоматически относятся к только что найденному элементу. Это

удобно, однако имя текущего каталога внутри подпрограммы обратного вызова отличается от начального каталога поиска.

Представьте, что нам надо с помощью модуля File::Find определить суммарный размер всех файлов, которые будут найдены. Процедура обратного вызова не может принимать входных аргументов, а вызывающая сторона – значение, возвращаемое подпрограммой. Но это не имеет никакого значения. После разыменования ссылки подпрограмма получает доступ ко всем видимым переменным. Например:

```
use File::Find;

my $total_size = 0;
find(sub { $total_size += -s if -f }, '.');
print $total_size, "\n";
```

Как и прежде, мы вызываем подпрограмму find и передаем ей два аргумента: ссылку на анонимную подпрограмму и имя начального каталога. При обнаружении файлов внутри этого каталога (и всех вложенных подкаталогах) она вызывает анонимную подпрограмму.

Обратите внимание: подпрограмма обращается к переменной $total_size. Мы объявили эту переменную за пределами области видимости подпрограммы find, однако она доступна в подпрограмме обратного вызова. Таким образом, даже при том, что подпрограмма find (которая не имеет прямого доступа к переменной $total_size) обращается к подпрограмме обратного вызова, последняя способна обращаться к переменной и изменять ее значение.

Подобные подпрограммы, которые имеют доступ ко всем переменным, существовавшим на момент объявления подпрограммы, называются *замыканиями* (термин заимствован из математики). В терминах языка Perl замыкание – это подпрограмма, которая может обращаться к лексическим переменным, расположенным вне области видимости данной подпрограммы.

Кроме того, возможность доступа к переменной изнутри замыкания гарантирует, что эта переменная будет существовать, по крайней мере до тех пор, пока существует ссылка на подпрограмму-замыкание. Попробуем подсчитать количество файлов:[1]

```
use File::Find;

my $callback;
{
    my $count = 0;
```

[1] На первый взгляд в этом отрывке имеется лишняя точка с запятой – в конце строки, где выполняется присваивание переменной $callback, не так ли? Постарайтесь запомнить: конструкция sub{...} – это выражение. Значение этого выражения (ссылка на подпрограмму) присваивается переменной $callback, поэтому в конце выражения должна стоять точка с запятой.

```
    $callback = sub { print ++$count, ": $File::Find::name\n" };
}
find($callback, '.');
```

Здесь объявляется переменная, которая будет хранить ссылку на подпрограмму обратного вызова. Эту переменную нельзя объявлять в пределах блока кода, потому что тогда Perl уничтожит ее после выхода из блока. Затем переменная `$count` инициализируется значением 0. После этого следует объявление анонимной подпрограммы, ссылка на которую записывается в переменную `$callback`. Данная подпрограмма представляет собой замыкание, поскольку она обращается к лексической переменной `$count`.

По выходе из блока кода переменная `$count` исчезает из области видимости. Однако, поскольку к ней производится обращение из подпрограммы, ссылка на которую продолжает оставаться в переменной `$callback`, переменная `$count` продолжает существовать в памяти как анонимная скалярная переменная.[1] При каждом вызове подпрограмма увеличивает значение переменной `$count`, получая в результате числа 1, 2, 3 и так далее.

Подпрограмма как возвращаемое значение другой подпрограммы

Блок кода прекрасно подходит для объявления подпрограммы обратного вызова, но гораздо удобнее иметь подпрограмму, которая возвращала бы ссылку на другую подпрограмму!

```
use File::Find;

sub create_find_callback_that_counts {
    my $count = 0;
    return sub { print ++$count, ": $File::Find::name\n" };
}

my $callback = create_find_callback_that_counts( );
find($callback, '.');
```

Здесь мы имеем практически то же, что и раньше, только оформили все несколько иначе. Подпрограмма `create_find_callback_that_counts()` инициализирует лексическую переменную `$count` значением 0 и возвращает ссылку на анонимную подпрограмму – тоже замыкание, поскольку

[1] Если точнее, то объявление замыкания увеличивает счетчик ссылок, как если бы на эту переменную явно была взята еще одна ссылка. Как раз перед концом блока счетчик ссылок на переменную `$count` имеет значение 2, а по завершении блока счетчик ссылок приобретает значение 1. Больше ниоткуда в программе нельзя обратиться к переменной `$count`, но она будет храниться в памяти до тех пор, пока будет существовать ссылка на подпрограмму в переменной `$callback` или где-либо еще.

ей доступна переменная $count. Эта переменная будет существовать в памяти даже после выхода из подпрограммы create_find_callback_that_counts() до тех пор, пока не исчезнет ссылка на анонимную подпрограмму.

При повторном обращении к той же подпрограмме обратного вызова переменная сохранит свое прежнее значение, поскольку переменная инициализируется в подпрограмме create_find_callback_that_counts(), а не в анонимной подпрограмме:

```
use File::Find;

sub create_find_callback_that_counts {
    my $count = 0;
    return sub { print ++$count, ": $File::Find::name\n" };
}

my $callback = create_find_callback_that_counts( );
print "мой каталог bin:\n";
find($callback, 'bin');
print "мой каталог lib:\n";
find($callback, 'lib');
```

Данный пример выведет последовательность чисел, начиная с 1, для всех файлов, которые будут найдены в каталоге bin, и затем продолжит эту последовательность, когда поиск будет производиться уже в каталоге lib. В обоих случаях задействуется одна и та же переменная, $count. Но если подпрограмма create_find_callback_that_counts() будет вызвана дважды, то мы получим две разных переменных $count:

```
use File::Find;

sub create_find_callback_that_counts {
    my $count = 0;
    return sub { print ++$count, ": $File::Find::name\n" };
}

my $callback1 = create_find_callback_that_counts( );
my $callback2 = create_find_callback_that_counts( );
print "мой каталог bin:\n";
find($callback1, 'bin');
print "мой каталог lib:\n";
find($callback2, 'lib');
```

В этом случае у нас появятся две разные переменные $count, каждая из которых будет доступна только из своей подпрограммы обратного вызова.

А как получить общее число файлов, подсчитанное подпрограммой обратного вызова? Ранее мы делали это с помощью глобальной переменной $total_size. Если объявить переменную $total_size в подпрограмме, которая возвращает ссылку на подпрограмму обратного вызова, мы не получим доступа к этой переменной. Но мы можем позволить себе пойти на небольшую хитрость. Мы можем определить, что в случае отсутствия входных аргументов подпрограмма обратного вызова

не должна ничего возвращать, но если она получает входной аргумент, то должна возвратить суммарный размер:

```
use File::Find;

sub create_find_callback_that_sums_the_size {
    my $total_size = 0;
    return sub {
        if (@_) { # это наш фиктивный запрос
            return $total_size;
        } else { # а это вызов из File::Find:
            $total_size += -s if -f;
        }
    };
}

my $callback = create_find_callback_that_sums_the_size( );
find($callback, 'bin');
my $total_size = $callback->('dummy'); # передача фиктивного аргумента,
                                       # чтобы получить суммарный размер
print "суммарный размер файлов в каталоге bin = $total_size\n";
```

Определение различной реакции на наличие и отсутствие входного аргумента – это далеко не универсальное решение. К счастью, мы можем вернуть более чем одну ссылку на подпрограмму из `create_find_callback_that_counts()`:

```
use File::Find;

sub create_find_callbacks_that_sum_the_size {
   my $total_size = 0;
   return(sub { $total_size += -s if -f }, sub { return $total_size });
}

my ($count_em, $get_results) = create_find_callbacks_that_sum_the_size( );
find($count_em, 'bin');
my $total_size = &$get_results( );
print " суммарный размер файлов в каталоге bin = $total_size\n";
```

Поскольку обе ссылки на подпрограммы были созданы в одной и той же области видимости, они обе имеют доступ к той же самой переменной `$total_size`. Даже при том что перед вызовом любой из этих подпрограмм переменная будет находиться за пределами области видимости, тем не менее обе подпрограммы будут иметь доступ к одной и той же переменной и могут быть использованы для получения результатов вычислений.

Возврат двух ссылок на подпрограммы не приводит к их вызову. Ссылки – это лишь данные, идентифицирующие точки вызова. Они будут исполняться, только когда вызываются как подпрограммы обратного вызова или в результате разыменования ссылок.

А что произойдет, если эта новая подпрограмма будет вызвана более чем один раз?

```
use File::Find;

sub create_find_callbacks_that_sum_the_size {
    my $total_size = 0;
    return(sub { $total_size += -s if -f }, sub { return $total_size });
}

## создание подпрограмм
my %subs;
foreach my $dir (qw(bin lib man)) {
    my ($callback, $getter) = create_find_callbacks_that_sum_the_size( );
    $subs{$dir}{CALLBACK} = $callback;
    $subs{$dir}{GETTER} = $getter;
}

## собрать данные
for (keys %subs) {
    find($subs{$_}{CALLBACK}, $_);
}

## вывести результаты
for (sort keys %subs) {
    my $sum = $subs{$_}{GETTER}->( );
    print "суммарный размер файлов в каталоге $_ = $sum bytes\n";
}
```

В разделе, где создаются подпрограммы, мы создали три экземпляра пар подпрограмм обратного вызова и получения результатов. Каждой подпрограмме обратного вызова соответствует подпрограмма получения результата. В следующем разделе выполняется сбор данных, здесь подпрограмма `find` вызывается три раза, причем каждый раз с другой ссылкой на подпрограмму обратного вызова. Благодаря этому результаты вычислений всякий раз записываются в свою переменную `$total_size`. В последнем разделе с помощью подпрограмм чтения производится вывод полученных результатов.

Каждая из шести подпрограмм (и каждая из трех переменных `$total_size`) имеет свой счетчик ссылок. Если мы попытаемся изменить содержимое хеша `%subs` или он выйдет из области видимости, это приведет к уменьшению счетчиков ссылок и утилизации занимаемой памяти. (Если эти элементы в свою очередь так же ссылаются на какие-либо другие данные, то их счетчики ссылок так же будут уменьшены соответствующим образом.)

Использование переменных замыканий для ввода данных

В предыдущих примерах было показано, как можно изменять переменные замыканий, но эти же переменные могут служить для передачи входной информации в замыкания. Попробуем написать подпрограмму, создающую подпрограмму обратного вызова для работы с мо-

дулем File::Find, которая в свою очередь будет выводить имена файлов с размером, превышающим некое значение:

```
use File::Find;

sub print_bigger_than {
    my $minimum_size = shift;
    return sub { print "$File::Find::name\n" if -f and -s >= $minimum_size };
}

my $bigger_than_1024 = print_bigger_than(1024);
find($bigger_than_1024, 'bin');
```

Здесь мы передаем подпрограмме print_bigger_than число 1024, которое затем переписывается в лексическую переменную $minimum_size. К этой переменной мы обращаемся из подпрограммы, на которую ссылается возвращаемое значение подпрограммы print_bigger_than, поэтому она становится переменной замыкания, значение которой сохраняется на протяжении всего жизненного цикла ссылки на подпрограмму. Как и прежде, каждый новый вызов подпрограммы print_bigger_than приводит к созданию нового экземпляра переменной $minimum_size, связанной с соответствующей ей ссылкой на подпрограмму.

Замыкания «замыкаются» только на лексических переменных, т. к. лексические переменные рано или поздно выйдут из области видимости. Поскольку глобальные переменные никогда не выйдут из области видимости, замыкания никогда не смогут замкнуться на них. Все подпрограммы всегда будут обращаться к одному и тому же экземпляру глобальной переменной.

Переменные замыканий как статические локальные переменные

Чтобы стать замыканием, подпрограмма не обязательно должна быть анонимной. Если именованная подпрограмма обращается к лексической переменной и эта переменная выйдет за пределы области видимости, подпрограмма сохранит возможность доступа ко всем своим лексическим переменным точно так же, как это происходит в случае анонимных подпрограмм. Рассмотрим две подпрограммы, которые ведут подсчет кокосовых орехов, собранных Джиллиганом:

```
{
    my $count;
    sub count_one { ++$count }
    sub count_so_far { return $count }
}
```

Если поместить этот отрывок в начало программы, то в результате будет создана переменная $count с областью видимости, ограниченной блоком кода, а две подпрограммы, которые ссылаются на эту переменную, превратятся в замыкания. Однако подпрограммы имеют опреде-

ленные имена, поэтому они продолжат свое существование и за пределами блока (как и любые другие именованные подпрограммы). А раз подпрограммы сохраняют возможность доступа к переменной даже за пределами области ее видимости, то они превращаются в замыкания и могут продолжать обращаться к переменной на протяжении всего жизненного цикла программы.

Убедиться в справедливости этого утверждения можно, выполнив несколько обращений к подпрограммам:

```
count_one( );
count_one( );
count_one( );
print 'мы собрали ', count_so_far( ), " орехов!\n";
```

Переменная `$count` сохраняет свое значение между вызовами подпрограмм `count_one()` или `count_so_far()`, и никакая другая часть программы не может получить доступ к этой переменной.

В языке C такие переменные известны как *статические локальные переменные*, то есть переменные, которые доступны только узкому кругу функций программы[1] и сохраняют свое значение на протяжении всего времени жизни программы между вызовами функций.

А что если нам потребуется вести счет в обратном порядке? Это можно сделать примерно так:

```
{
    my $countdown = 10;
    sub count_down { $countdown-- }
    sub count_remaining { $countdown }
}

count_down( );
count_down( );
count_down( );
print 'нам осталось собрать ', count_remaining( ), " орехов!\n";
```

Данный прием эффективен только в том случае, если этот блок будет размещаться где-нибудь ближе к началу программы, то есть до того, как будет вызвана функция `count_down()` или `count_remaining()`. Догадываетесь почему?

Прием не сработает, если блок будет размещаться после вызова любой из функций. Причина проста: первая строка этого блока функционально делится на две части:

```
my $countdown = 10;
```

[1] Если быть совсем точными, то *статические локальные переменные* C/C++ доступны внутри только одной функции, в которой они определены; переменные, доступные узкому кругу функций, – это *статические переменные файла*; и те и другие обладают особенностями, на которых акцентирует внимание автор. – *Примеч. науч. ред.*

Первая часть – это объявление лексической переменной `$countdown`. Эта часть обрабатывается на *этапе компиляции*. Вторая часть – это присвоение переменной значения 10. Эта часть обрабатывается уже на *этапе исполнения*. Пока вторая часть строки не пройдет обработку на этапе исполнения, переменная `$countdown` будет иметь значение по умолчанию, а именно `undef`.

В качестве одного из решений этой проблемы можно порекомендовать преобразовать блок со статической переменной в блок `BEGIN`:

```
BEGIN {
    my $countdown = 10;
    sub count_down { $countdown-- }
    sub count_remaining { $countdown }
}
```

Ключевое слово `BEGIN` сообщает компилятору Perl, что после компиляции блока необходимо на время приостановить компиляцию программы и исполнить этот блок. Если в процессе исполнения блока не возникло фатальной ошибки, компиляция программы будет продолжена со строки, стоящей сразу за блоком. Кроме того, вслед за этим сам блок удаляется из программы, благодаря чему гарантируется, что он будет исполнен всего один раз, даже если синтаксически он находится внутри цикла или подпрограммы.

Упражнения

Ответы на эти вопросы вы найдете в приложении A в разделе «Ответы к главе 7».

Упражнение 1 [50 мин]

Профессор обновил некоторые файлы в понедельник днем и, к несчастью, забыл, какие именно. Это случается с ним постоянно. Он хочет, чтобы вы написали подпрограмму с именем `gather_mtime_between`, которой можно было бы передать начальную и конечную дату и время временного интервала и которая возвращала бы две ссылки на подпрограммы. Первая из них должна с помощью модуля `File::Find` отыскать все файлы, которые были изменены в течение указанного интервала, а вторая должна возвращать список найденных файлов.

Ниже приводится пример программного кода, который вы можете опробовать. Он отыскивает только те файлы, которые были изменены в последний понедельник, хотя его нетрудно переориентировать на любой другой день недели. (Весь этот текст не надо вводить вручную, поскольку программа в виде файла с именем `ex6-1.plx` входит в состав архива с примерами к книге, который можно загрузить с веб-сайта издательства O'Reilly.)

Подсказка: время последнего изменения файла (`mtime`) можно определить, например, так:

```
my $timestamp = (stat $file_name)[9];
```

Поскольку это часть массива, круглые скобки обязательны. Не забывайте, что имя рабочего каталога, внутри которого вызывается подпрограмма обратного вызова, не обязательно будет совпадать с именем начального каталога, которое было передано подпрограмме `find`.

```
use File::Find;
use Time::Local;

my $target_dow = 1; # воскресенье = 0, понедельник = 1, ...
my @starting_directories = (".");

my $seconds_per_day = 24 * 60 * 60;
my($sec, $min, $hour, $day, $mon, $yr, $dow) = localtime;

my $start = timelocal(0, 0, 0, $day, $mon, $yr); # сегодня в полночь
while ($dow != $target_dow) {
    # Назад на одни сутки
    $start -= $seconds_per_day; # надеюсь, что не попал на момент перехода
                                # между летним временем и зимним! :-)
    if (--$dow < 0) {
        $dow += 7;
    }
}
my $stop = $start + $seconds_per_day;

my($gather, $yield) = gather_mtime_between($start, $stop);
find($gather, @starting_directories);
my @files = $yield->( );

for my $file (@files) {
    my $mtime = (stat $file)[9]; # получение времени последнего изменения
    my $when = localtime $mtime;
    print "$when: $file\n";
}
```

Обратите внимание на комментарий по поводу перехода между летним временем и зимним. Во многих странах осуществляется переход с летнего времени на зимнее и обратно. В день перевода часов длина суток уже не будет равна 86400 секундам. В данном примере эта проблема не учитывается, однако вы, как более педантичный программист, могли бы принять ее во внимание.

8

Ссылки на дескрипторы файлов

Мы рассмотрели возможность передачи массивов, хешей и подпрограмм по ссылке, ввели дополнительные уровни абстракции для разрешения некоторых типов проблем. Однако кроме этого ссылки могут служить для хранения дескрипторов файлов. Взглянем еще раз на старые проблемы и новые пути их решения.

Старый способ

В былые времена дескрипторы файлов обозначались именами, не имевшими специального префикса. Дескриптор файла – это еще один тип данных в языке Perl, хотя об этом не вспоминают уже достаточно давно, поскольку он не имеет специального обозначения. Вам, вероятно, уже встречался программный код с подобными именами дескрипторов.

```
open LOG_FH, '>> castaways.log'
        or die "Невозможно открыть файл castaways.log: $!";
```

Что произойдет, если передать подобный дескриптор в другую часть программы, например библиотечной подпрограмме? Наверное, вам приходилось видеть и несколько более загадочный код, в котором фигурируют значения специального типа `typeglob`:

```
log_message( *LOG_FH, 'Путешественники высадились на берег!' );

log_message( *LOG_FH, 'Астронавт выдержал перегрузки' );
```

В подпрограмме `log_message()` выбирается первый элемент из списка аргументов и сохраняется в другой переменной типа `typeglob`. Не вдаваясь в детали, заметим лишь, что переменные типа `typeglob` хранят указатели на все переменные пакета, имеющие то же самое имя. Когда присваивается значение одной переменной типа `typeglob` другой переменной того же типа, создается псевдоним, по которому можно будет обращаться к тем же самым данным, включая и дескрипторы файлов.

Затем, когда это имя будет выступать в качестве дескриптора файла, Perl уже будет знать, что необходимо выбрать ту часть переменной типа typeglob, которая является дескриптором файла. Было бы намного проще, если бы дескрипторы файлов имели специальное обозначение!

```
sub log_message {
    local *FH = shift;

    print FH @_, "\n";
}
```

Обратите внимание на спецификатор local. Для работы с переменными типа typeglob используется таблица символов, то есть операции выполняются с переменными пакета. Переменные пакета не являются лексическими переменными, поэтому спецификатор my здесь недопустим. Таким образом, чтобы избежать возможных конфликтов имен из-за того, что где-то в пакете может быть объявлена другая переменная с именем FH, необходим спецификатор local. Данный спецификатор говорит, что значение переменной FH временное, что оно будет действовать только внутри подпрограммы log_message и что после завершения подпрограммы Perl должен восстановить прежнее значение переменной FH, если таковая вообще существовала.

Если все это вам надоело и вы не хотите об этом думать, ну и ладно! Есть способ и получше. Будем считать, что этот раздел вы даже не видели, – переходите к чтению следующего.

Улучшенный способ

Начиная с Perl 5.6 функция open получила возможность создавать ссылку на дескриптор файла в виде обычного скалярного значения. Теперь мы можем обозначать дескрипторы файлов не именами без специального префикса, а обычными скалярными переменными со значением по умолчанию undef.

```
my $log_fh;
open $log_fh, '>> castaways.log'
        or die "Невозможно открыть файл castaways.log: $!";
```

Из этого ничего не выйдет, если скалярная переменная уже имеет какое-либо значение, потому что Perl не сможет идентифицировать данную переменную как дескриптор файла. Значение $log_fh будет интерпретироваться как символическая ссылка, в результате печать будет произведена в дескриптор файла с именем 5. При наличии директивы use strict это приведет к фатальной ошибке.

```
my $log_fh = 5;
open $log_fh, '>> castaways.log'
        or die "Невозможно открыть файл castaways.log: $!";
print $log_fh "Нам нужно больше орехов!\n"; # не работает
```

Главный принцип Perl заключается в том, чтобы все делать за один шаг. Мы можем объявить переменную прямо в операторе open. Снача-

ла это будет казаться странным, но через пару (ладно, чуть больше) раз вы привыкнете, потому что ко всему хорошему привыкаешь быстро.

```
open my $log_fh, '>> castaways.log'
        or die "Невозможно открыть файл castaways.log: $!";
```

При выводе данных в файл вместо прежнего обозначения дескриптора файла мы можем аналогичным образом задействовать скалярную переменную. Обратите внимание: здесь после дескриптора файла попрежнему нет запятой.

```
print $log_fh "Сегодня у нас нет бананов!\n";
```

Такой синтаксис может показаться вам забавным, но даже если нет, он может показаться несколько странным человеку, который позднее будет просматривать вашу программу. В книге «Best Perl Practices» Дэмиан Конвей (Damian Conway) рекомендует окружать переменную дескриптора файла фигурными скобками, чтобы явно указать ее предназначение. Такой синтаксис больше напоминает операторы `grep` и `map` со встроенными блоками.

```
print {$log_fh} "Сегодня у нас нет бананов!\n";
```

Теперь мы можем работать с дескрипторами файлов как с обычными скалярными переменными. И для этого нам не понадобятся какие-либо хитрости или уловки.

```
log_message( $log_fh, 'Меня зовут мистер Эд' );

sub log_message {
    my $fh = shift;
    print $fh @_, "\n";
}
```

Кроме того, мы можем создавать ссылки на дескрипторы файлов, чтобы потом читать данные из этих файла. Для этого достаточно передать имя файла во втором аргументе.

```
open my $fh, "castaways.log"
        or die "Невозможно открыть файл castaways.log: $!";
```

После чего получаем возможность использовать в операторе ввода скалярную переменную вместо прежнего обозначения дескриптора файла. Ранее нам пришлось бы вставить между угловыми скобками имя дескриптора без префикса:

```
while( <LOG_FH> ) { ... }
```

Теперь же его можно заменить скалярной переменной.

```
while( <$log_fh> ) { ... }
```

Вообще везде, где раньше дескрипторы файлов обозначались именем без префикса, теперь можно использовать скалярные переменные со ссылками.

В любой из этих форм записи, когда скалярная переменная выходит из области видимости (или когда ей присваивается какое-либо другое значение), Perl закрывает файл. Нам не надо самим закрывать файл.

Способ еще лучше

До сих пор в наших примерах демонстрировалась форма обращения к оператору `open` с двумя аргументами, но на самом деле их может быть больше: во втором аргументе вместе с именем файла можно дополнительно указать режим открытия. Таким образом, получается, что в одной строке передаются два разных параметра, и мы должны свято верить в возможности Perl, чтобы понять и принять такую форму записи.

Чтобы избежать возможной путаницы, мы разобьем второй аргумент на два отдельных.

```
open my $log_fh, '>>', 'castaways.log'
        or die "Невозможно открыть файл castaways.log: $!";
```

Такая форма записи с тремя аргументами обладает дополнительным преимуществом доступа к фильтрам ввода-вывода Perl. Мы не будем здесь слишком углубляться в обсуждение.[1] В `perlfunc` описание функции `open` занимает более 400 строк, и это при том, что для нее имеется специальный раздел в справочном руководстве `perldoc perlopentut`.

IO::Handle

Для выполнения всех необходимых действий с дескрипторами файлов в Perl предназначен модуль `IO::Handle`, работающий «за кулисами». Таким образом, наш скаляр на самом деле является объектом.[2] Пакет `IO::Handle` представляет собой основной класс, посредством которого выполняются операции ввода-вывода, то есть его функции намного шире, чем простое управление файлами.

Вам едва ли придется напрямую иметь дело с модулем `IO::Handle`, если только вы не создаете собственные модули ввода-вывода. Ему лучше предпочесть более удобные модули, работающие поверх `IO::Handle`. Мы еще не обсуждали тему объектно-ориентированного программирования (ООП); ее черед наступит в главе 11. Поэтому пока руководствуйтесь примерам из документации к модулю.

---

[1] Брайан написал статью «Get More Out of Open», которая увидела свет в журнале «The Perl Journal» 31 октября 2005 года, *http://www.tpj.com/documents/s=9923/tpj1130955178261/bdf_open.htm*.

[2] Приходилось ли вам когда-нибудь задаваться вопросом, почему после имени дескриптора файла не ставится запятая? В действительности это косвенное обозначение объекта (о чем мы еще не упоминали, если, конечно, вы не прочитали книгу прежде, чем стали обращать внимание на сноски в соответствии с нашими рекомендациями!).

Функции некоторых из этих модулей могут выполняться встроенной функцией open (в зависимости от версии Perl). Но для выполнения операций ввода-вывода гораздо удобнее модули. В этом случае вместо встроенной функции open мы можем задействовать интерфейс модуля. Тогда впоследствии, чтобы изменить поведение программы, достаточно будет изменить имя подключаемого модуля. А поскольку мы ориентируемся на использование интерфейса модуля, изменение имени – это не такая уж сложная работа.

IO::File

Модуль IO::File наследует функциональные возможности модуля IO::Handle и предназначен специально для работы с файлами. Модуль входит в состав стандартного дистрибутива Perl и потому уже должен иметься у вас. Создать объект IO::File можно несколькими способами.

Создать ссылку на дескриптор файла можно с помощью конструктора, передав ему единственный аргумент. В успехе выполнения операции можно убедиться, проверив значение переменной-ссылки.

```
use IO::File;

my $fh = IO::File->new( '> castaways.log' )
        or die "Невозможно открыть файл castaways.log: $!";
```

Если вам не нравится такая форма записи (по тем же причинам, что и форма обращения к обычной функции open), можно воспользоваться другой формой. Режим открытия файла можно указать во втором необязательном аргументе.[1]

```
my $read_fh = IO::File->new( 'castaways.log', 'r' );

my $write_fh = IO::File->new( 'castaways.log', 'w' );
```

Режим открытия файла можно определить точнее с помощью битовой маски. Для этих целей модуль IO::File предоставляет дополнительные константы.

```
my $append_fh = IO::File->new( 'castaways.log', O_WRONLY|O_APPEND );
```

Кроме обычного именованного файла, модуль IO::File позволяет открыть неименованный временный файл. Если операционная система допускает подобную возможность, чтобы получить дескриптор файла, открытого для записи и чтения, то достаточно создать новый объект.

```
my $temp_fh = IO::File->new_tmpfile;
```

Такого рода файлы закрываются автоматически, когда программа покидает область видимости переменной. Если этого недостаточно, файл можно закрыть явно.

---

1 Эти строки режимов открытия файла определены в стандарте ANSI C для функции fopen. Аналогичный формат представления режима допускается в функции open, поскольку в действительности она использует объект IO::File.

```
$temp_fh->close;

undef $append_fh;
```

Анонимные объекты IO::File

Если не хранить объекты IO::File в скалярных переменных, то для выполнения некоторых операций придется прибегнуть к несколько иному синтаксису. Предположим, что необходимо скопировать все файлы, имена которых согласуются с шаблоном *.input, в соответствующие им файлы с расширением .output, причем выполнить все операции копирования необходимо одновременно. В этом случае в первую очередь необходимо открыть все входные и выходные файлы:

```
my @handlepairs;

foreach my $file ( glob( '*.input' ) ) {
    (my $out = $file) =~ s/\.input$/.output/;
    push @handlepairs, [
         (IO::File->new("<$file") || die),
         (IO::File->new(">$out") || die),
    ];
}
```

Мы получили массив ссылок на массивы, каждый элемент которых представляет собой объект IO::File. Теперь можно копировать данные.

```
while (@handlepairs) {
    @handlepairs = grep {
        if (defined(my $line = $_->[0]->getline)) {
            print { $_->[1] } $line;
          } else {
            0;
        }
    } @handlepairs;
}
```

Здесь пары входных и выходных файлов передаются оператору grep, имеющему следующую структуру:

```
@handlepairs = grep { CONDITION } @handlepairs;
```

На каждом проходе в списке остаются только те пары дескрипторов, которые в результате проверки условия CONDITION дают значение «истина». Внутри блока условия производится попытка чтения из первого элемента пары. Если операция завершилась успехом, прочитанная строка записывается во второй элемент пары (дескриптор соответствующего выходного файла). Если операция записи прошла успешно, возвращается значение «истина», и тогда оператор grep оставляет данную пару в списке. Если операция записи или чтения терпит неудачу, оператор grep получит в качестве результата проверки условия значение «ложь» и удалит пару из списка. Удаление пары дескрипторов из списка автоматически приводит к закрытию обоих файлов. Отлично!

Обратите внимание: мы не можем обратиться к более привычным операциям чтения и записи из/в дескрипторы, потому что мы отказались от простых скалярных переменных. Со скалярными переменными операции чтения-записи реализуются гораздо проще:

```
while (@handlepairs) {
    @handlepairs = grep {
        my ($IN, $OUT) = @$_;
        if (defined(my $line = <$IN>)) {
            print $OUT $line;
          } else {
            0;
        }
    } @handlepairs;
}
```

Такой алгоритм, по нашему мнению, более предпочтителен, поскольку действия с простыми скалярными переменными воспринимаются человеком легче, чем операции со сложными ссылками. Кроме того, можно избавиться от оператора if в цикле:

```
while (@handlepairs) {
    @handlepairs = grep {
        my ($IN, $OUT) = @$_;
        my $line;
        defined($line = <IN>) and print $OUT $line;
      } @handlepairs;
}
```

Если вы понимаете, что оператор and учитывает результат вычисления предыдущей части выражения и что функция print возвращает значение «истина» только в том случае, когда все в порядке, то такая замена может считаться вполне оправданной. Помните лозунг языка Perl: «Любое действие можно сделать несколькими способами» (хотя не все они одинаково хороши или правильны).

IO::Scalar

Иногда возникает необходимость записать данные не в файл, а в строку. Интерфейсы некоторых модулей не предоставляют такой возможности, поэтому мы должны реализовать алгоритм работы программы таким образом, чтобы он напоминал вывод данных в файл через дескриптор файла. Подобная потребность может быть обусловлена необходимостью зашифровать данные перед записью в файл, сжать их или отправить данные по электронной почте прямо из программы.

Модуль IO::Scalar предоставляет возможность работы со скалярами через дескриптор файла посредством оператора tie. Этот модуль отсутствует в стандартном дистрибутиве Perl, поэтому его придется установить отдельно.

```
use IO::Scalar;
```

```
my $string_log = '';
my $scalar_fh = IO::Scalar->new( \$string_log );

print $scalar_fh "Частный клуб Хоуэлла закрыт\n";
```

Теперь наши сообщения будут попадать в скалярную переменную `$string_log`, а не в файл. А как быть, если возникнет необходимость прочитать данные? Сделать это можно точно так же, как и в случае с обычным файлом. В примере, который приводится ниже, сначала создается дескриптор `$scalar_fh`, как и в предыдущем примере. Затем выполняется операция чтения из строки. В цикле `while` извлекаются строки, содержащие имя Джиллиган (которых в журнале наверняка будет больше всего, т. к. он всегда попадает в какие-нибудь неприятности).

```
use IO::Scalar;

my $string_log = '';
my $scalar_fh = IO::Scalar->new( \$string_log );

while( <$scalar_fh> ) {
        next unless /'Джиллиган'/;
        print;
        }
```

Начиная с версии Perl 5.8 то же самое можно сделать, не прибегая к услугам модуля `IO::Scalar`.

```
open( my $fh, '>>', \$string_log )
        or die "Невозможно открыть строку для дополнения! $!";
```

IO::Tee

Как быть, если информацию необходимо одновременно вывести более чем в одно место? Что надо сделать, чтобы, например, записать некоторые данные в файл и в строку одновременно? Опираясь на уже имеющиеся у нас знания, мы могли бы написать что-нибудь вроде:

```
my $string = '';

open my $log_fh, '>>', 'castaways.log'
    or die "Невозможно открыть файл castaways.log;
open my $scalar_fh, '>>', \$string;

my $log_message = "Minnow спущен на воду!\n"
print $log_fh    $log_message;
print $scalar_fh $log_message;
```

Конечно же, мы могли бы несколько сократить объем программного кода и обойтись всего одним обращением к оператору `print`. Для этого можно было бы в цикле `foreach` выполнить обход всех ссылок на дескрипторы файлов, поочередно записывать их в промежуточную переменную `$fh` и выводить данные с ее помощью.

```
foreach my $fh ( $log_fh, $scalar_fh ) {
```

```
        print $fh $log_message;
    }
```

Однако при таком подходе нам все равно приходится проделывать довольно много работы. Так, при добавлении цикла foreach нам пришлось выбирать, какие дескрипторы включать в список. Существует ли возможность определить группу дескрипторов под одним общим названием и производить вывод в эту группу? Да, такую возможность дает модуль IO::Tee. Представьте себе тройник на водопроводной трубе. Когда вода доходит до тройника, она направляется по двум ответвлениям одновременно. Когда данные выводятся средствами модуля IO::Tee, они могут быть отправлены в два (или более) разных канала одновременно. Таким образом, IO::Tee мультиплексирует вывод. В нашем примере сообщения должны попадать в файл журнала и в скалярную переменную.

```
use IO::Tee;

$tee_fh = IO::Tee->new( $log_fh, $scalar_fh );

print $tee_fh "Радио продолжает работать даже посреди океана!\n";
```

Но это еще не все. Если первым аргументом модулю IO::Tee передается дескриптор ввода (все последующие аргументы обязательно должны быть дескрипторами вывода), мы сможем использовать получившийся групповой дескриптор как для чтения, так и для записи. Канал ввода и канал вывода – это разные вещи, но благодаря IO::Tee мы получаем возможность работать с ними через единственный дескриптор.

```
use IO::Tee;

$tee_fh = IO::Tee->new( $read_fh, $log_fh, $scalar_fh );

# прочитать данные из $read_fh
my $message = <$tee_fh>;

# вывести данные в $log_fh и $scalar_fh
print $tee_fh $message;
```

Дескриптор $read_fh не обязательно должен быть связан с файлом. Это может быть открытый сокет, скалярная переменная, вывод внешней команды[1] или что-то другое, что только вам заблагорассудится.

Ссылки на дескрипторы каталогов

Так же, как ссылки на дескрипторы файлов, могут быть созданы ссылки на дескрипторы каталогов.

```
opendir my $dh, '.' or die "Невозможно открыть каталог: $!";

foreach my $file ( readdir( $dh ) ) {
```

[1] Создать дескриптор канала для внешней команды, доступный для чтения, можно с помощью модуля IO::Pipe.

```
        print "Шкипер, я нашел $file!\n";
    }
```

Работа со ссылками на каталоги выполняется в соответствии с теми же правилами, о которых мы рассказали выше. Со ссылкой можно работать, только если скалярная переменная не имеет определенного значения, и дескриптор автоматически закрывается, если программа покидает область видимости переменной или когда переменной присваивается какое-либо значение.

IO::Dir

Для работы с дескрипторами каталогов также можно использовать объектно-ориентированные интерфейсы. Модуль `IO::Dir` вошел в состав стандартного дистрибутива Perl начиная с версии 5.6. Он не несет в себе новой функциональности, а представляет собой лишь обертку вокруг некоторых встроенных функций языка Perl.[1]

```
use IO::Dir;

my $dir_fh = IO::Dir->new( '.' ) || die "Невозможно открыть каталог! $!\n";

while( defined( my $file = $dir_fh->read ) ) {
       print "Шкипер, я нашел $file!\n";
       }
```

Нам не надо создавать дескриптор каталога повторно, если позднее в этой же программе мы решим просмотреть содержимое каталога еще раз. Нам достаточно будет переустановить его в начало:

```
while( defined( my $file = $dir_fh->read ) ) {
       print "Я нашел $file!\n";
       }

# спустя некоторое время
$dir_fh->rewind;

while( defined( my $file = $dir_fh->read ) ) {
       print "Я опять нашел $file!\n";
       }
```

Упражнения

Ответы на эти вопросы вы найдете в приложении A в разделе «Ответы к главе 8».

Упражнение 1 [20 мин]

Напишите программу, которая будет выводить число, месяц, год и день недели, но при этом она должна дать пользователю возможность выби-

---

[1] Имя каждого метода `IO::Dir` начинается с приставки `dir`, а описания методов имеются в справочном руководстве `perlfunc`.

рать, записывать ли данные в файл, в скалярную переменную или одновременно и в файл, и в переменную. Независимо от того, какой вариант выберет пользователь, вывод должен производиться единственным обращением к функции `print`. Если пользователь предпочтет вывод в скалярную переменную, программа должна выводить ее содержимое на экран перед завершением работы.

Упражнение 2 [30 мин]

Профессору надо прочитать файл журнала, который выглядит следующим образом:

```
Джиллиган: 1 кокос
Шкипер: 3 кокоса
Джиллиган: 1 банан
Джинжер: 2 папайи
Профессор: 3 кокоса
МэриЭнн: 2 папайи
...
```

Он хочет создать несколько файлов с именами `Джиллиган.info`, `МэриЭнн.info` и т. д. В каждый из этих файлов из журнала должны переписываться строки, начинающиеся с имени персонажа. (Имена всегда отделяются от оставшейся части строки символом двоеточия.) Например, файл `Джиллиган.info` должен начинаться со строк:

```
Джиллиган: 1 кокос
Джиллиган: 1 банан
```

Сейчас файл журнала достаточно велик, а компьютер на кокосовом процессоре не отличается высокой скоростью, поэтому Профессор хотел бы, чтобы программа выполняла всего один проход журнала и выводила бы информацию во все файлы параллельно.

Подсказка: используйте хеш, ключами которого были бы имена потерпевших кораблекрушение, а значениями – объекты `IO::File`, которые могут создаваться по мере необходимости.

Упражнение 3 [15 мин]

Напишите программу, которая принимала бы из командной строки имена нескольких каталогов и выводила бы их содержимое. Используйте функцию, которая принимает в качестве аргумента ссылку на дескриптор каталога, созданную с помощью `IO::Dir`.

9

Практические приемы работы со ссылками

В этой главе мы рассмотрим оптимизацию сортировки и обработку рекурсивно определенных данных.

Краткий обзор способов сортировки

Встроенная функция `sort` в языке Perl по умолчанию сортирует текстовые строки в алфавитном порядке.[1] Она замечательно справляется со строками:

```
my @sorted = sort qw(Джиллиган Шкипер Профессор Джинджер Мэри_Энн);
```

но если ей передать список чисел:

```
my @wrongly_sorted = sort 1, 2, 4, 8, 16, 32;
```

в результате мы получим такую последовательность: `1`, `16`, `2`, `32`, `4`, `8`. Почему функция сортировки дала неправильный порядок? Просто потому, что эта функция интерпретирует элементы списка как строки и сортирует их в соответствии с правилами, применяемыми при сортировке строк. Любая строка, начинающаяся с символа `3` должна стоять раньше, чем какая-либо строка, начинающаяся с символа `4`.

Если порядок сортировки, принятый по умолчанию, вас не устраивает, это вовсе не означает, что вам придется полностью реализовать весь

[1] Мои друзья называют его «ASCII-витный» порядок сортировки. Обычно язык Perl использует для сортировки не коды ASCII, а заданный по умолчанию порядок сортировки, который зависит от текущих языковых настроек и используемого набора символов. Более подробно об этом можно прочитать на странице справочного руководства `man perllocale` (не `perllocal!`).

алгоритм сортировки, поскольку все необходимое уже имеется в Perl. Но какой бы алгоритм вы ни использовали, вам все равно придется сравнивать пары элементов A и B и решать, какой из них должен следовать первым. Именно эту часть алгоритма сортировки вам придется написать, все остальное Perl сделает сам.

По умолчанию при выполнении сортировки Perl сравнивает элементы списка как строки. Мы можем определить иной способ сравнения, вставив *блок сортировки* между оператором sort и списком элементов.[1] Внутри блока сортировки существуют переменные $a и $b, которые содержат значения двух элементов списка. Если мы будем сортировать числа, то в $a и $b будут находиться два числа из списка.

Чтобы указать, в каком порядке должны располагаться текущие два элемента, блок сортировки должен возвращать результат сравнения в определенном виде. Если элемент $a должен предшествовать элементу $b, должно возвращаться число -1. Если элемент $b должен предшествовать элементу $a, должно возвращаться число 1. Если порядок следования не имеет значения, должно возвращаться число 0. Порядок следования может оказаться неважным, например при сравнении двух слов «Фред» и «ФРЕД» без учета регистра символов или при сравнении одинаковых чисел, например 42 и 42.[2]

Попробуем отсортировать числа в надлежащем порядке, для чего напишем свой блок сортировки, в котором будут сравниваться элементы $a и $b:

```
my @numerically_sorted = sort {
    if ($a < $b)    { -1 }
    elsif ($a > $b) { +1 }
    else            { 0 }
  } 1, 2, 4, 8, 16, 32;
```

Теперь элементы списка сравниваются как числа, поэтому в результате мы получим правильный порядок следования. Конечно, при таком способе приходится вводить с клавиатуры довольно много кода, но мы можем сократить его, применив оператор <=>:

```
my @numerically_sorted = sort { $a <=> $b } 1, 2, 4, 8, 16, 32;
```

---

1 Здесь можно также вставить обращение к подпрограмме, которая будет вызываться для сравнения пар элементов.

2 На самом деле вместо чисел -1 и +1 можно возвращать любые отрицательные или положительные числа соответственно. Современные версии Perl обладают *устойчивым* механизмом сортировки. Так, если блок сортировки возвращает нулевое значение при сравнении элементов $a и $b, то гарантируется, что на выходе порядок следования этих элементов будет соответствовать оригинальному. Старые версии Perl таких гарантий дать не могли, возможно, и в будущих версиях устойчивый механизм сортировки использоваться не будет, поэтому не стоит на это полагаться.

Оператор <=> сравнивает элементы и возвращает значения –1, 0 или +1 в соответствии с правилами, которые мы только что описали. Сортировка в обратном порядке в Perl выполняется очень просто:[1]

```
my @numerically_descending =
    reverse sort { $a <=> $b } 1, 2, 4, 8, 16, 32;
```

Однако это не единственный способ сортировки чисел в обратном порядке. Оператор <=> ничего не знает о существовании списка и понятия не имеет, откуда взялись значения параметров $a и $b. Он видит только, что одно значение стоит слева от него, а другое справа. Поменяв параметры $a и $b местами, мы получим обратный порядок сортировки:

```
my @numerically_descending =
    sort { $b <=> $a } 1, 2, 4, 8, 16, 32;
```

Если раньше при сравнении пары элементов возвращалось значение -1, то теперь это выражение будет возвращать значение +1, и наоборот. Таким образом, список будет отсортирован в обратном порядке, и при этом нам не потребовался оператор reverse. Этот прием легко запомнить: если $a стоит слева, а $b – справа, то первым в списке будет стоять наименьший элемент (точно так же, как стояли бы символы a и b в отсортированном списке).

Какой из способов лучше? Когда следует предпочесть оператор reverse, а когда надо менять местами элементы $a и $b? В большинстве случаев эффективность этих двух способов практически одинакова, поэтому в первую очередь выбор следует делать исходя из соображений удобочитаемости программного кода и отдавать предпочтение оператору reverse. Однако в более сложных случаях одного только оператора reverse может оказаться недостаточно.

У оператора <=> есть коллега – оператор cmp, сортирующий строки, хотя на практике он применяется редко, потому что функция sort() по умолчанию сортирует элементы как строки. Чаще всего оператор cmp нужен для реализации более сложных алгоритмов сравнения, которые мы вскоре продемонстрируем.

Сортировка по индексам

В главе 2 при помощи индексов мы преодолели некоторые трудности в операторах grep и map. Аналогичным образом индексы можно задействовать в реализации алгоритмов сортировки, дающих довольно интересные результаты. Попробуем отсортировать список имен персонажей:

```
my @sorted = sort qw(Джиллиган Шкипер Профессор Джинджер Мэри_Энн);
print "@sorted\n";
```

[1] Начиная с версии 5.8.6 Perl правильно интерпретирует последовательность операторов reverse sort и сразу выполняет сортировку в обратном порядке, не создавая промежуточный временный список.

В результате мы получим следующий список:[1]

```
Джиллиган Джинджер Мэри_Энн Профессор Шкипер
```

Но как быть, если потребуется узнать, какой элемент исходного списка стал первым, вторым, третьим и т. д. элементом отсортированного списка? Например, имя Джинджер занимает вторую позицию в отсортированном списке, а в исходном списке оно занимало четвертую позицию. Как узнать, что второй элемент получившегося списка был четвертым элементом в исходном?

Тут нет ничего сложного. Достаточно перейти от сортировки имен к сортировке их индексов:

```
my @input = qw(Джиллиган Шкипер Профессор Джинджер Мэри_Энн);
my @sorted_positions = sort { $input[$a] cmp $input[$b] } 0..$#input;
print "@sorted_positions\n";
```

В данном случае `$a` и `$b` представляют не элементы списка, а их индексы. Таким образом, вместо того чтобы сравнивать значения `$a` и `$b`, мы сравниваем `$input[$a]` и `$input[$b]` как строки с помощью оператора `cmp`. В результате индексы сортируются в соответствии с алфавитным порядком следования имен, которые они представляют. В данном примере мы получили последовательность индексов `0 3 4 2 1`. Это означает, что первый элемент оригинального списка остался на первой позиции в отсортированном. Второй элемент отсортированного списка занимал третью позицию в оригинальном и т. д. Теперь мы можем не только отсортировать список, но и проанализировать полученные результаты.

Мы знаем, какую позицию занимал элемент в оригинальном списке, но пока еще не знаем, какую позицию займет элемент после сортировки. Следующий пример дает ответ на этот вопрос:

```
my @input = qw(Джиллиган Шкипер Профессор Джинджер Мэри_Энн);
my @sorted_positions = sort { $input[$a] cmp $input[$b] } 0..$#input;
my @ranks;
@ranks[@sorted_positions] = (0..$#sorted_positions);
print "@ranks\n";
```

Этот пример выведет последовательность чисел `0 4 3 1 2`. Это означает, что позиция имени Джиллиган после сортировки не изменится, имя Шкипер переместится в позицию с номером `4`, Профессор – в позицию с номером `3` и т. д. В данном случае нумерация позиций начинается с 0, поэтому, чтобы перейти к нумерации, более понятной для обычного человека, нужно к полученным номерам добавить единицу. Для этого вместо `0..@sorted_positions` можно использовать `1..@sorted_positions`, например так:

```
my @input = qw(Джиллиган Шкипер Профессор Джинджер Мэри_Энн);
```

[1] И он именно таким получается для переведенных на русский язык элементов списка (в ActivePerl 5.8.8.). – *Примеч. науч. ред.*

```
my @sorted_positions = sort { $input[$a] cmp $input[$b] } 0..$#input;
my @ranks;

@ranks[@sorted_positions] = (1..@sorted_positions);
for (0..$#ranks) {
    print "Имя $input[$_] переместится в позицию $ranks[$_]\n";
}
```

Результат работы этого примера:

```
Имя Джиллиган переместится в позицию 1
Имя Шкипер переместится в позицию 5
Имя Профессор переместится в позицию 4
Имя Джинджер переместится в позицию 2
Имя Мэри_Энн переместится в позицию 3
```

Данная методика может оказаться достаточно удобной везде, где необходимо проанализировать данные с разных точек зрения. Например, когда порядок следования элементов данных из соображений эффективности определяется порядком следования некоторых числовых индексов, но время от времени возникает необходимость вывести их в алфавитном порядке. Или при анализе системных журналов серверов, когда выполнить сортировку по некоторым элементам, например по номеру месяца, бывает не так просто.

Эффективность алгоритмов сортировки

Профессор отвечает за поддержание в исправном состоянии компьютерной техники (собранной целиком из бамбука, кокосовых орехов, ананасов и работающей под управлением обезьянки, обученной языку Perl), поэтому он все время обнаруживает, что люди оставляют слишком много данных в единственной обезьяньей файловой системе, и поэтому решил распечатать список недобросовестных пользователей.

Профессор написал подпрограмму `ask_monkey_about()`. Ей передается имя одного из потерпевших кораблекрушение, а она возвращает объем пространства в ананасах, которое данный человек занял для хранения своей информации. Самый простой способ вывести список пользователей в порядке убывания расходуемого ими пространства выглядит примерно так:

```
my @castaways =
    qw(Джиллиган Шкипер Профессор Джинджер Мэри_Энн Торстон Ловей);
my @wasters = sort {
    ask_monkey_about($b) <=> ask_monkey_about($a)
  } @castaways;
```

Теоретически такой метод вполне работоспособен. Для первой пары имен (Джиллиган и Шкипер) мы спрашиваем у обезьянки: «Сколько ананасов занял Джиллиган?» и «Сколько ананасов занял Шкипер?». Обратно получаем два числа, с помощью которых определяем порядок следования Джиллигана и Шкипера в заключительном списке.

Однако наступает момент, когда необходимо сравнить число ананасов, занятое Джиллиганом и другим его сотоварищем. Например, предположим, что следующая пара – это Джинджер и Джиллиган. Мы спрашиваем у обезьянки о Джинджер, получаем от нее ответ и затем спрашиваем о Джиллигане... снова. Это наверняка не понравится обезьянке, поскольку о Джиллигане мы уже спрашивали. Но этот алгоритм заставит нас запрашивать каждое значение по два, три или даже четыре раза, чтобы разместить всего семь значений в требуемом порядке.

Это обстоятельство может сильно испортить нам жизнь, потому что обезьянка выйдет из себя.

Как же свести к минимуму число вопросов, которые должны быть заданы обезьянке? Для начала, например, можно построить таблицу. Мы будем передавать оператору map список из семи имен, а на выходе получать семь ссылок на массивы, каждый из которых будет содержать имя потерпевшего кораблекрушение и количество ананасов, которое сообщит обезьянка.

```
my @names_and_pineapples = map {
    [ $_, ask_monkey_about($_) ]
  } @castaways;
```

Здесь мы задали обезьянке семь вопросов. На этом наше общение с ней должно быть закончено! У нас теперь есть все, что нужно, чтобы решить поставленную задачу.

На следующем шаге нам необходимо отсортировать ссылки на массивы, упорядочив их по значениям, которые нам сообщила обезьянка:

```
my @sorted_names_and_pineapples = sort {
    $b->[1] <=> $a->[1];
  } @names_and_pineapples;
```

В данной подпрограмме значения $a и $b – это два элемента списка, подлежащего сортировке. Когда $a и $b представляют числа, мы сортируем их как числа, когда $a и $b представляют ссылки, мы должны сортировать их как ссылки, для этого мы разыменовываем их и извлекаем из массивов элементы с индексом 1 (количество ананасов, которое нам сообщила обезьянка). Поскольку $b стоит слева от $a, сортировка будет выполняться в порядке убывания. (Профессор хочет, чтобы самые недобросовестные пользователи стояли в самом начале списка.)

На этом работу можно считать законченной. А если нам потребуется только список имен без количества ананасов, то достаточно преобразовать ссылки в массив имен с помощью еще одного оператора map:

```
my @names = map $_->[0], @sorted_names_and_pineapples;
```

Каждый элемент первоначального списка будет попадать в переменную $_, затем эта переменная будет разыменована и из полученного массива будет выбран элемент с индексом 0, который содержит только имя персонажа.

Теперь у нас есть список имен, отсортированный в порядке убывания количества занимаемых ананасов, который нам удалось получить всего за три простых шага.

Преобразование Шварца

Промежуточные переменные в этом примере больше нигде не встречаются, они служат только для того, чтобы подготовить исходные данные для очередного шага. Мы можем упростить решение задачи, объединив все шаги:

```
my @names =
    map $_->[0],
    sort { $b->[1] <=> $a->[1] }
    map [ $_, ask_monkey_about($_) ],
    @castaways;
```

Поскольку операторы `map` и `sort` выполняются справа налево, мы должны читать эту конструкцию снизу вверх. Итак, сначала мы берем список `@castaways`, создаем ссылки на массивы, задавая обезьянке простые вопросы, сортируем список ссылок и затем извлекаем имена из каждой ссылки на массив. В результате мы получаем список имен, отсортированный в требуемом порядке.

Эту конструкцию обычно называют *преобразованием Шварца* в честь Рэндала, пославшего много лет тому назад сообщение в Usenet. С той поры преобразование Шварца не раз доказывало свою эффективность и заслуживает, чтобы каждый программист положил его в свою копилку практических приемов сортировки.

Если это преобразование покажется вам сложным для запоминания или вы хотите понять основные его принципы, вам поможет следующая схема, в которой мы выделили постоянные и изменяемые части:

```
my @output_data =
    map $_->[0],
    sort { СРАВНЕНИЕ ПАР ЭЛЕМЕНТОВ $a->[1] И $b->[1] }
    map [ $_, ВЫЗОВ ДОРОГОСТОЯЩЕЙ ФУНКЦИИ ОТ $_ ],
    @input_data;
```

В основе преобразования лежит отображение первоначального списка в список ссылок на массивы, дорогостоящий вызов функции для каждого элемента списка производится всего один раз. Сортировка ссылок на массивы производится на основе значений, полученных в результате обращения к этой функции.[1] После этого извлекаются первоначальные значения, но уже в новом порядке. Чтобы воспользоваться этим преобразованием, нам достаточно реализовать две операции. Ниже

[1] Дорогостоящей называется такая операция, выполнение которой занимает значительное время или в процессе ее исполнения потребляется значительный объем памяти.

приводится программный код, реализующий сортировку строк без учета регистра символов на основе преобразования Шварца:[1]

```
my @output_data =
    map $_->[0],
    sort { $a->[1] cmp $b->[1] }
    map [ $_, "\U$_" ],
    @input_data;
```

Многоуровневая сортировка на основе преобразования Шварца

Если необходимо отсортировать данные в соответствии с несколькими критериями одновременно, мы можем воспользоваться преобразованием Шварца.

```
my @output_data =
    map $_->[0],
    sort { СРАВНЕНИЕ ПАР ЭЛЕМЕНТОВ $a->[1] И $b->[1] or
           СРАВНЕНИЕ ПАР ЭЛЕМЕНТОВ $a->[2] И $b->[2] ПО ДРУГОМУ КРИТЕРИЮ or
           СРАВНЕНИЕ ПАР ЭЛЕМЕНТОВ $a->[2] И $b->[2] ПО ТРЕТЬЕМУ КРИТЕРИЮ }
map [ $_, НЕКОТОРАЯ ФУНКЦИЯ ОТ $_, ЕЩЕ ОДНА ФУНКЦИЯ, И ЕЩЕ ОДНА ],
@input_data;
```

В этом шаблоне предусмотрена сортировка по трем различным критериям на основе трех значений, вычисленных и записанных в анонимные массивы (вместе с оригинальными данными, которые должны быть отсортированы и которые всегда записываются в первый элемент массива).

Данные с рекурсивной организацией

До этого момента мы приводили примеры обработки данных, имеющих фиксированную структуру, но в реальной жизни нередко приходится обслуживать данные с иерархической структурой, которая определяется рекурсивно.

Первый пример: таблица HTML, состоящая из строк и ячеек, которые могут в свою очередь содержать целые таблицы. Второй пример: иерархия файловой системы, содержащей каталоги, которые в свою очередь содержат файлы и вложенные подкаталоги. Третий пример: организационная структура какой-либо компании, состоящей из нескольких

---

[1] Этот способ оказывается эффективным только в случае высокой стоимости операции приведения символов к верхнему регистру, что наблюдается для очень длинных строк или когда количество строк очень велико. Для небольшого числа коротких строк более эффективным может оказаться такой способ: `my @output_data = sort { "\U$a" cmp "|U$b" } @input_data`. Если вы сомневаетесь, попробуйте сравнить производительность этих двух методов.

подразделений, каждое из которых в свою очередь может делиться на более мелкие подразделения. И четвертый пример: очень сложная организационная диаграмма, которая может содержать экземпляры таблиц HTML из первого примера, элементы файловой системы из второго примера и даже целые организационные диаграммы…

Для получения, хранения и обработки информации с такой иерархической структурой мы можем использовать ссылки. Зачастую подпрограммы, предназначенные для обработки таких структур данных, реализуют рекурсивные алгоритмы.

Рекурсивные алгоритмы позволяют вести обработку данных с неограниченной степенью сложности, начиная с некоторого базового случая.[1] Базовый случай определяет порядок действий, которые необходимо выполнить в простейшем случае: когда лист дерева не имеет ветвей, когда массив пуст или когда счетчик достиг нулевого значения. Фактически каждая из ветвей рекурсивного алгоритма должна иметь свой базовый случай. Рекурсивный алгоритм без базового случая – это бесконечный цикл.[2]

Рекурсивные подпрограммы обязательно имеют ветвь, из которой они вызывают сами себя для обработки очередного уровня вложенности данных, и ветвь, из которой не происходит рекурсивный вызов – здесь обрабатывается базовый случай. В случае с первым примером, который приводился выше, базовым случаем будет пустая ячейка таблицы. В качестве базовых случаев можно также рассматривать пустые строки таблицы и пустые таблицы. Для второго примера базовыми случаями будут файлы и пустые каталоги.

Например, рекурсивная подпрограмма, вычисляющая факториал числа, которая является простейшей рекурсивной функцией, может выглядеть так:

```
sub factorial {
    my $n = shift;
    if ($n <= 1) {
        return 1;
      } else {
        return $n * factorial($n - 1);
    }
}
```

[1] При применении рекурсивных алгоритмов всегда должен иметься элементарный базовый случай, для вычисления которого не требуется рекурсивный вызов функции, а все остальные рекурсивные вызовы должны в конечном счете приводить к этому случаю, иначе рекурсивная функция просто зациклится.

[2] Собственно, не бесконечный цикл, а рекурсия бесконечной глубины, которая, скорее всего, закончится «переполнением стека» и аварийным завершением задачи. – *Примеч. науч. ред.*

Здесь мы имеем две ветви рекурсивного алгоритма: базовый случай, когда значение переменной `$n` меньше или равно 1 и рекурсивный вызов подпрограммы не производится, и рекурсивный случай, когда значение переменной `$n` больше 1 и производится рекурсивный вызов подпрограммы для обработки очередного уровня вложенности данных (то есть вычисление факториала числа, меньшего на 1, чем текущее).

Эта задача может быть решена более эффективно при помощи итерационного алгоритма, хотя факториал чаще определяется как рекурсивная операция.[1]

Построение структур данных с рекурсивной организацией

Предположим, что нам необходимо получить сведения о файловой системе, включая имена файлов и имена каталогов со всем вложенным в них содержимым. Представьте себе каталог в виде хеша, в котором ключами являются имена файлов и каталогов, а значениями: для простых файлов – `undef`, а для каталогов – ссылки на другие хеши. В этом случае пример представления каталога `/bin` может выглядеть так:

```
my $bin_directory = {
    cat  => undef,
    cp   => undef,
    date => undef,
    ... и так далее ...
};
```

В домашнем каталоге Шкипера тоже может находиться персональный каталог `bin`, в котором находятся его программы:

```
my $skipper_bin = {
    navigate            => undef,
    discipline_gilligan => undef,
    eat                 => undef,
};
```

В этой структуре нет никакой информации о местонахождении данного каталога в иерархии файловой системы. Она просто отражает структуру некоторого каталога.

Теперь поднимемся на один уровень вверх – к домашнему каталогу Шкипера, в котором, скорее всего, вместе с каталогом `bin` будут содержаться и другие файлы:

```
my $skipper_home = {
    '.cshrc'                    => undef,
    'Please_rescue_us.pdf'      => undef,
```

[1] Для этого разработана формальная техника преобразования рекурсивных алгоритмов в итерационные. – *Примеч. науч. ред.*

```
    'Things_I_should_have_packed' => undef,
    bin                           => $skipper_bin,
};
```

Обратите внимание, что здесь у нас четыре элемента: три файла и четвертый элемент bin, значение которого не равно undef – это ссылка на хеш, созданный ранее и описывающий каталог bin Шкипера. Именно так выглядят подкаталоги в нашей структуре данных. Если значение элемента хеша равно undef, значит, это файл, но если значением элемента является ссылка на хеш, то это вложенный подкаталог, в котором могут находиться другие файлы и подкаталоги. Разумеется, два определения, приведенные выше, можно объединить:

```
my $skipper_home = {
    '.cshrc'                    => undef,
    Please_rescue_us.pdf        => undef,
    Things_I_should_have_packed => undef,

    bin => {
        navigate            => undef,
        discipline_gilligan => undef,
        eat                 => undef,
    },
};
```

Теперь иерархическая природа данных стала более наглядной.

Очевидно, описывать такие структуры данных вручную дело не из легких. Поэтому лучше создать для этой цели подпрограммы. Попробуем написать подпрограмму, которая будет возвращать undef, если заданное имя в файловой системе является файлом, и ссылку на хеш с содержимым каталога, если заданное имя является каталогом. Базовый случай, когда мы встречаем файл, самый простой в реализации, поэтому начнем с него:

```
sub data_for_path {
    my $path = shift;
    if (-f $path) {
        return undef;
    }
    if (-d $path) {
        ...
    }
    warn "Имя $path не является ни файлом, ни каталогом\n";
    return undef;
}
```

Если эта подпрограмма будет вызвана для файла .cshrc в домашнем каталоге Шкипера, она вернет значение undef, показывая, что это файл.

Теперь перейдем к той части подпрограммы, которая обрабатывает подкаталоги. В первую очередь необходимо создать хеш. Затем для каждо-

го из элементов хеша рекурсивным вызовом подпрограммы определяются их значения:

```
sub data_for_path {
    my $path = shift;
    if (-f $path or -l $path) { # файлы или символические ссылки
        return undef;
    }
    if (-d $path) {
        my %directory;
        opendir PATH, $path or die "Невозможно открыть $path: $!";
        my @names = readdir PATH;
        closedir PATH;
        for my $name (@names) {
            next if $name eq '.' or $name eq '..';
            $directory{$name} = data_for_path("$path/$name");
        }
        return \%directory;
    }
    warn "Имя $path не является ни файлом, ни каталогом\n";
    return undef;
}
```

Базовыми случаями в этом рекурсивном алгоритме являются файлы и символические ссылки. Этот алгоритм не смог бы корректно воссоздать структуру файловой системы, если бы следовал за символическими ссылками на каталоги, а также если бы ссылки были жесткими, потому что мог бы попасть в бесконечный цикл, если бы символическая ссылка указывала на каталог, в котором она находится.[1] Этот алгоритм также оказался бы бессильным перед файловой системой с некорректной структурой, например когда каталоги образуют не древовидную, а циклическую структуру. Хотя ошибки в структуре файловой системы встречаются не так часто, но вообще говоря, рекурсивные алгоритмы весьма чувствительны к любым нарушениям в данных с рекурсивной организацией.

Значение каждого элемента каталога определяется с помощью рекурсивного вызова подпрограммы `data_for_path`. Для файлов это будет значение `undef`. Когда возвращается ссылка на хеш, она превращается в ссылку на анонимный хеш, поскольку после выхода из подпрограммы имя хеша выйдет из области видимости. (Сами данные при этом не изменяются, зато изменяется число способов, которыми к ним можно обратиться.)

---

[1] Каждый из нас в свое время пытался сделать нечто подобное, а потом мучился в поисках ответа на вопрос, почему программа зациклилась. Как бы то ни было, но когда с этой проблемой мы столкнулись во второй раз, это уже не было нашей ошибкой, а в третий раз это была всего лишь досадная случайность. Это наш опыт, и мы хотим поделиться им.

Если в каталоге будет обнаружен вложенный подкаталог, рекурсивный вызов подпрограммы извлечет содержимое каталога с помощью `readdir` и вернет ссылку на хеш, которая будет вставлена в хеш, созданный вызывающей подпрограммой.

На первый взгляд рекурсивный алгоритм может показаться чем-то мистическим, но если пройтись по нему более внимательно, мистика исчезнет. Проверим результаты работы подпрограммы, передав ей имя каталога `.` (текущий каталог):

```
use Data::Dumper;
print Dumper(data_for_path('.'));
```

Совершенно очевидно, что результаты будут выглядеть гораздо интереснее, если каталог содержит вложенные подкаталоги.

Отображение данных с рекурсивной организацией

Подпрограмма `Dumper` модуля `Data::Dumper` выводит данные в достаточно удобочитаемом формате, но как быть, если нам необходимо будет вывести данные в несколько ином формате? Можно написать свою подпрограмму. Как и в случае создания данных с рекурсивной организацией, подпрограмма, которая выводит эти данные, обычно тоже рекурсивная.

Чтобы вывести данные, нам необходимо знать имя каталога, представляющего вершину дерева, поскольку его имя не хранится в самой структуре:

```
sub dump_data_for_path {
    my $path = shift;
    my $data = shift;

    if (not defined $data) { # обычный файл
        print "$path\n";
        return;
    }
    ...
}
```

В случае с простым файлом мы выводим его полное имя. Когда `$data` представляет ссылку на хеш, мы должны обойти все его ключи и вывести их значения:

```
sub dump_data_for_path {
    my $path = shift;
    my $data = shift;

    if (not defined $data) { # обычный файл
        print "$path\n";
        return;
    }
```

```
        my %directory = %$data;

        for (sort keys %directory) {
            dump_data_for_path("$path/$_", $directory{$_});
        }
    }
```

Мы передаем подпрограмме полный путь к каждому элементу каталога, полученный в результате объединения имени вмещающего каталога с именем текущего элемента, и указатель, содержащий значение undef для обычного файла или ссылку на хеш в случае вложенного подкаталога. Результаты можно увидеть, запустив команду:

```
dump_data_for_path('.', data_for_path('.'));
```

Опять же результаты будут выглядеть гораздо интереснее, если каталог будет содержать вложенные подкаталоги, а полученный вывод будет напоминать результаты работы команды:

```
find . -print
```

запущенной из командной оболочки.

Упражнения

Ответы на эти вопросы вы найдете в приложении A в разделе «Ответы к главе 9».

Упражнение 1 [15 мин]

Элементарная сортировка элементов каталога /bin по их размерам может быть выполнена с помощью оператора glob:

```
my @sorted = sort { -s $a <=> -s $b } glob "/bin/*";
```

Перепишите этот пример с применением преобразования Шварца.

Если в вашем каталоге /bin недостаточно большое число файлов, скорее всего вы пользуетесь операционной системой, отличной от UNIX. В этом случае передайте оператору glob имя какого-либо другого каталога.

Упражнение 2 [15 мин]

Ознакомьтесь с описанием модуля Benchmark, входящего в состав дистрибутива Perl. Напишите программу, с помощью которой можно было бы ответить на вопрос: «Насколько эффективнее оказалась реализация примера из первого упражнения на основе преобразования Шварца?»

Упражнение 3 [10 мин]

С помощью алгоритма преобразования Шварца прочитайте список слов и отсортируйте их в алфавитном порядке. При этом не учитываются регистры символов и знаки пунктуации. Подсказка: обратите внимание на следующий пример, который может оказаться полезным:

```
my $string = 'Мэри-Энн';
$string =~ tr/А-Я/а-я/; # перевести символы в нижний регистр
$string =~ tr/а-я//cd;  # удалить все символы вне диапазона а-я
print $string;          # выведет "мэриэнн"
```

Программа не должна изменять исходные данные! Если на вход подаются имена `Профессор` и `шкипер`, они должны выводиться точно в том виде, в каком были поданы на вход (в том же порядке с учетом регистра символов).

Упражнение 4 [20 мин]

Измените подпрограмму рекурсивного вывода содержимого каталога так, чтобы имена вложенных каталогов отображались с отступами. Пустые каталоги должны отображаться как:

```
sandbar, пустой каталог
```

а непустые каталоги должны выводиться со всем своим содержимым, с отступом в два пробела:

```
uss_minnow, with contents:
  anchor
  broken_radio
  galley, with contents:
    captain_crunch_cereal
    gallon_of_milk
    tuna_fish_sandwich
  life_preservers
```

10

Разработка больших программ

В этой главе рассказано, как поделить программу на отдельные блоки, а также о некоторых трудностях, возникающих при объединении частей в одну программу, например когда отдельные блоки создаются разными разработчиками.

Ликвидация повторяющихся участков программного кода

Шкипером было написано множество программ, имеющих целью обеспечить возможность навигации по всем известным портам, куда заходил «Minnow». Он обнаружил, что в каждой его программе участвует одна и та же подпрограмма:

```
sub turn_toward_heading {
    my $new_heading = shift;
    my $current_heading = current_heading( );
    print "Текущий курс ", $current_heading, ".\n";
    print "Требуемый курс $new_heading ";
    my $direction = 'право';
    my $turn = ($new_heading - $current_heading) % 360;
    if ($turn > 180) { # поворот влево будет короче
        $turn = 360 - $turn;
        $direction = 'лево';
    }
    print ", $direction руля на $turn градусов.\n";
}
```

Данная подпрограмма вычисляет кратчайший угол поворота от текущего курса корабля (возвращается подпрограммой `current_heading()`) до требуемого (передается подпрограмме в виде первого аргумента).

Первую строку подпрограммы можно было бы заменить:

```
my ($new_heading) = @_;
```

Это самый распространенный способ извлечения аргумента. В любом случае первый аргумент записывается в переменную `$new_heading`. Однако первый способ, при котором элемент удаляется из `@_`, имеет свои преимущества, в чем мы скоро убедимся. Поэтому будем придерживаться первого способа разбора входных аргументов. А сейчас вернемся к обсуждаемой теме...

Написав с десяток программ, где используется эта подпрограмма, Шкипер пришел к мнению, что она выводит совершенно бессмысленные сообщения, например когда корабль уже идет требуемым курсом (или просто дрейфует в нужном направлении). В конце концов, если предположить, что в настоящий момент корабль идет курсом 234° и требуемый курс также составляет 234°, то подпрограмма выводит:

```
Текущий курс 234.
Требуемый курс 234, право руля на 0 градусов.
```

Эдак кто угодно устанет! Шкипер решил исправить этот недостаток, добавив проверку величины угла поворота на равенство нулю:

```
sub turn_toward_heading {
    my $new_heading = shift;
    my $current_heading = current_heading( );
    print "Текущий курс ", $current_heading, ".\n";
    my $direction = 'право';
    my $turn = ($new_heading - $current_heading) % 360;
    unless ($turn) {
        print "Так держать (отличная работа!).\n";
        return;
    }
    print "Требуемый курс $new_heading ";
    if ($turn > 180) { # поворот влево будет короче
        $turn = 360 - $turn;
        $direction = 'лево';
    }
    print ", $direction руля на $turn градусов.\n";
}
```

Отлично. Новая подпрограмма замечательно работает в текущей программе навигации. Но, как мы уже отмечали, эта подпрограмма вставлена и в другие программы, которые продолжат действовать Шкиперу на нервы своими бессмысленными сообщениями.

Шкиперу надо придумать такой способ, чтобы каждая из подпрограмм существовала в единственном экземпляре и совместно могла бы использоваться несколькими программами. Как обычно, Perl позволяет достичь этого несколькими способами.

Вставка программного кода с помощью eval

Шкипер сможет сэкономить дисковое пространство (и клетки своего серого вещества), поместив определение подпрограммы turn_toward_heading в отдельный файл. Допустим, что Шкипер выявил полдюжины подпрограмм общего пользования, связанных с управлением корабля, которые используются, если не во всех, то, по крайней мере, в большинстве написанных им программ. Он может перенести их в отдельный файл с именем *navigation.pm*, где будут находиться только необходимые подпрограммы.

Но как после этого сообщить Perl о том, что подпрограммы находятся в другом файле? Самый сложный способ заключается в использовании функции eval, о которой мы уже говорили в главе 2.

```
sub load_common_subroutines {
    open MORE_CODE, 'navigation.pm' or die "navigation.pm: $!";
    undef $/; # прочитать файл целиком
    my $more_code = <MORE_CODE>;
    close MORE_CODE;
    eval $more_code;
    die $@ if $@;
}
```

Perl считывает содержимое файла navigation.pm в переменную $more_code. После этого текст, содержащийся в переменной, интерпретируется с помощью функции eval. Все лексические переменные в $more_code останутся локальными по отношению к интерпретируемому блоку программного кода.[1] Если в процессе интерпретации будут обнаружены какие-либо синтаксические ошибки, Perl установит значение переменной $@ и аварийно завершит работу программы с соответствующим сообщением.

Теперь вместо того чтобы вставлять несколько десятков строк подпрограмм общего пользования в каждую программу, достаточно вставить единственную подпрограмму в каждый из файлов.

Однако такой подход нельзя назвать оптимальным, если нам придется продолжать решать проблему данным способом. К счастью, в арсенале Perl имеются методы, которые помогут решить эту проблему.

С помощью оператора do

Шкипер переместил подпрограммы общего пользования в отдельный файл navigation.pm. Если теперь Шкипер вставит строки:

---

[1] Как это ни странно, переменная $more_code также будет видна из этого блока кода, однако это едва ли имеет какое-то практическое значение с точки зрения изменения программного кода в ходе интерпретации.

```
do 'navigation.pm';
die $@ if $@;
```

во все свои программы, это будет равноценно вызову функции `eval` для содержимого файла в данной точке программы.[1]

Таким образом, оператор `do` действует так, как если бы программный код из файла `navigation.pm` был вставлен в текущую программу. Правда, в виде отдельного блока со своей областью видимости, поэтому лексические переменные (переменные, объявленные со спецификатором `my`) и большинство директив (таких как `use strict`), объявленные в подключаемом файле, не видны в главной программе.

Теперь Шкипер может спокойно поддерживать единственную копию подпрограмм общего пользования без необходимости копировать все изменения и дополнения в множество разнообразных программ, которые были написаны и используются им. На рис. 10.1 показано, как можно работать с такой общей библиотекой.

Разумеется, такой подход требует соблюдения определенных соглашений, поскольку изменение интерфейса той или иной подпрограммы

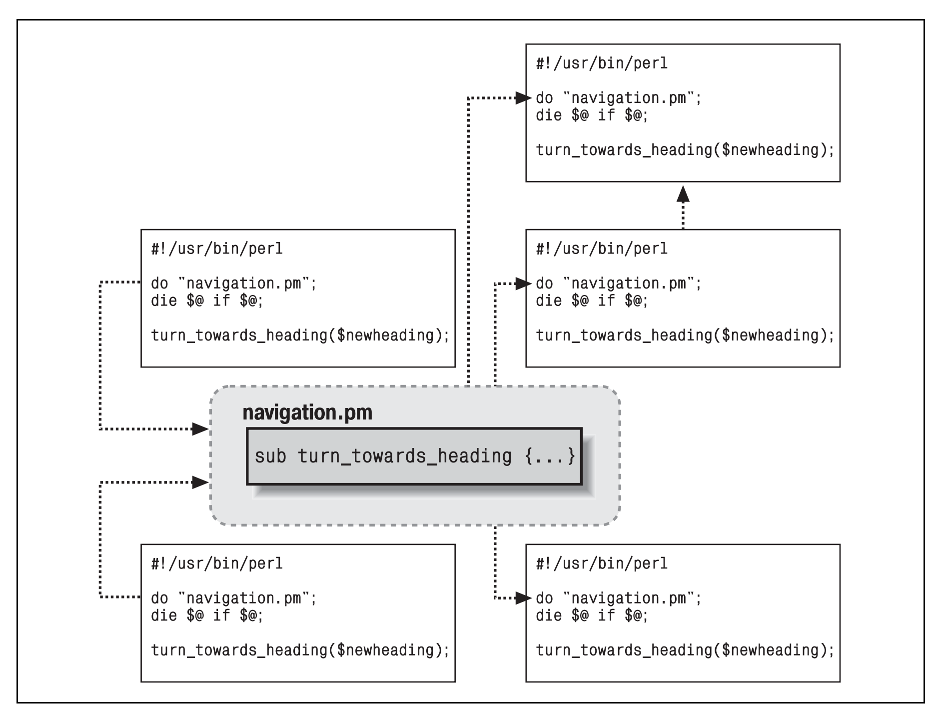


Рис. 10.1. *Файл navigation.pm задействован во всех программах, написанных Шкипером*

[1] Данный способ не влияет на `@INC`, `%INC` и не обрабатывает ситуацию отсутствия файла, о чем мы вскоре поговорим.

может оказать отрицательное влияние сразу на многие программы.[1] Теперь Шкипер должен с особым тщанием проработать строение программных компонентов многократного пользования и как следует продумать модульную архитектуру. Будем считать, что Шкипер приобрел некоторый опыт, но мы еще вернемся к этой теме в следующих главах.

После того как программный код будет оформлен в виде отдельного файла, им смогут воспользоваться другие программисты, и наоборот. Если Джиллиган напишет подпрограмму `drop_anchor()` и поместит ее в файл `drop_anchor.pm`, Шкипер сможет использовать ее, подключив библиотеку Джиллигана:

```
do 'drop_anchor.pm';
die $@ if $@;
...
drop_anchor( ) if at_dock( ) or in_port( );
```

Такая организация программного кода упрощает его обслуживание и облегчает взаимодействие между программистами.

Если программный код в файлах `.pm` содержит только определения подпрограмм, то проще всего, пожалуй, загружать эти подпрограммы с помощью оператора `do`.

Теперь вернемся на секунду к библиотеке `drop_anchor.pm` и представим, что Шкипер пишет программу, которая не только вычисляет изменение курса, но и «бросает якорь»:

```
do 'drop_anchor.pm';
die $@ if $@;
do 'navigation.pm';
die $@ if $@;
...
turn_toward_heading(90);
...
drop_anchor( ) if at_dock( );
```

Все работает просто замечательно. Подпрограммы, описанные в обеих библиотеках, стали доступны основной программе.

С помощью директивы require

Предположим, что подпрограммы из `navigation.pm` также обращаются к подпрограмме, описанной в `drop_anchor.pm`. Сначала Perl выполнит чтение файла прямо из программы, а затем еще раз во время анализа пакета `navigation`. Вследствие этого произойдет ненужное нам переопределение подпрограммы `drop_anchor()`. Хуже того, если при запуске

[1] В последующих главах мы покажем, как создаются тесты, предназначенные для проверки программного кода общего пользования.

программы будет разрешен вывод предупреждений,[1] то Perl сообщит о том, что функция была переопределена, даже если это одно и то же определение.

Таким образом, нам необходим некий механизм, который следил бы за тем, какие файлы были загружены, чтобы не загружать их повторно. В языке Perl для этих целей предусмотрена директива `require`. Предыдущий программный код поменяйте на следующий:

```
require 'drop_anchor.pm';
require 'navigation.pm';
```

Директива `require` будет следить за всеми загружаемыми файлами.[2] Загрузив и обработав файл один раз, Perl будет игнорировать все последующие попытки загрузить его с помощью директивы `require`. Это означает, что даже если в файле `navigation.pm` будет стоять директива `require "drop_ancor.pm"`, Perl загрузит файл `drop_ancor.pm` только один раз и избавит нас от надоедливых предупреждений о повторном переопределении подпрограмм (рис. 10.2). И самое главное – это поможет нам сэкономить время, затрачиваемое на повторную загрузку файла и анализ его содержимого.

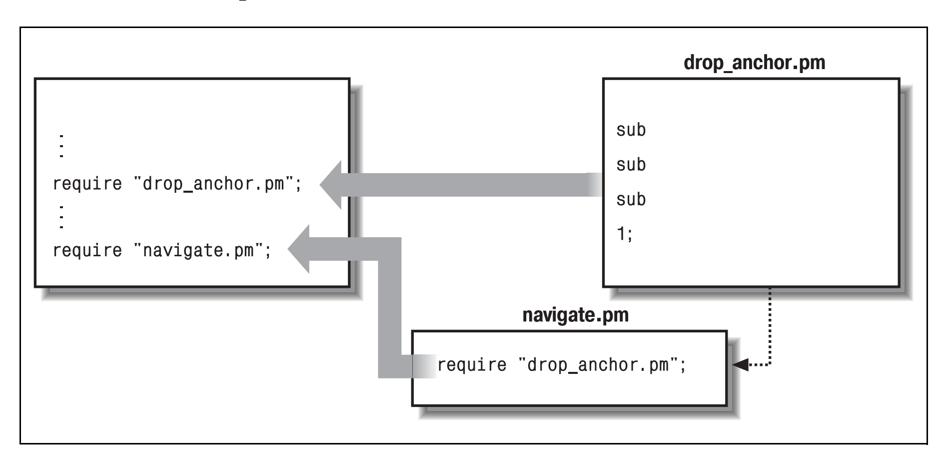


Рис. 10.2. *Как только Perl загрузит файл drop_anchor.pm, он будет игнорировать другие попытки загрузить его*

Кроме того, директива `require` обладает двумя дополнительными особенностями:

- Если в процессе интерпретации файла, загружаемого с помощью директивы `require`, будут обнаружены синтаксические ошибки, Perl

[1] Ну, вы-то *разрешаете* вывод предупреждений? Вывод предупреждений включается с помощью `-w` или `warnings`.

[2] С помощью хеша `%INC`, о чем говорится в описании директивы `require` в справочнике `perlfunc`.

аварийно завершит работу программы, что избавляет от необходимости вставлять дополнительную проверку `die $@ if $@`.

- Последнее выражение в файле, которое вычисляется в процессе интерпретации, должно возвращать значение «истина».

Из-за второго пункта большинство файлов, подключаемых с помощью директивы `require`, содержат в конце странную строку с числом 1. Тем самым гарантируется, что последним вычисленным выражением в файле будет «истина». Постарайтесь следовать этой устоявшейся традиции.

Изначально значение «истина» представляло собой способ сообщить вызывающей программе, что программный код был благополучно проанализирован и что он не содержит ошибок. Однако подавляющее большинство программистов настолько привыкло к стратегии `die if ...`, что стратегия «значение последнего выражения – «ложь» ныне рассматривается как историческое недоразумение.

require и @INC

До сих пор мы ничего не говорили о структуре каталогов, где размещаются файлы с основной программой и библиотеками. Дело в том, что в простейшем случае, когда программа расположена там же, где и подключаемые библиотеки, и запускается оттуда же, «все работает и так».

Ситуация усложняется, если библиотеки размещаются в другом каталоге. Фактически Perl ищет файлы библиотек в предопределенном списке каталогов поиска – примерно так же, как командная оболочка производит поиск файлов с помощью переменной окружения `PATH`. Текущий каталог (в ОС UNIX это каталог с именем `.` (точка)) является одним из элементов в списке каталогов поиска. Таким образом, если библиотеки находятся в текущем каталоге, все будет работать без проблем.

Список каталогов поиска – это список элементов в специальном массиве `@INC`, о чем мы уже коротко упоминали в главе 3. По умолчанию массив содержит в себе имя текущего каталога и еще примерно с полдесятка каталогов, которые были определены во время сборки Perl. Введя команду `perl -V`, в последних нескольких строках вы получите список каталогов. А команда позволит получить только список каталогов `@INC`.[1]

```
perl -le 'print for @INC'
```

Если вы не являетесь тем человеком, который несет ответственность за работоспособность Perl, то скорее всего не будете иметь права записи в эти каталоги, за исключением текущего. В этом случае все эти каталоги должны быть доступны вам для записи. В них Perl ищет библиотеки и модули, общие для всей системы.

[1] В командной строке Windows одинарные кавычки следует заменить двойными.

Дополнение списка @INC

Вполне вероятно, что мы не сможем установить дополнительные модули в один из предопределенных каталогов из списка @INC. Но у нас есть возможность дополнить этот список своими каталогами до обращения к директиве require, что позволит Perl произвести поиск и в наших каталогах. Массив @INC — это обычный массив; поэтому если Шкипер пожелает добавить в него свой каталог, он может вставить в программу, например, такую строку:

```
unshift @INC, '/home/skipper/perl-lib';
```

Теперь к списку каталогов поиска в дополнение к предопределенным будет добавлен каталог с библиотеками Шкипера. Поиск в первую очередь выполняется в нем, поскольку это первый элемент в массиве @INC. Если каталог добавляется к списку оператором unshift, а не push, Perl автоматически устраняет возможные конфликты имен между файлами Шкипера и библиотеками, установленными в системе, отдавая приоритет библиотекам Шкипера.

Обычно список каталогов дополняется еще до того, как будет сделано что-либо еще, поэтому данное действие, как правило, помещается внутри блока BEGIN и исполняется на этапе компиляции, то есть до того, как на этапе исполнения отработает какая-либо директива require. В противном случае Perl исполнит операции в той последовательности, в какой они расположены в тексте программы, и тогда надо убедиться, что оператор unshift предшествует директиве require.

```
BEGIN {
    unshift @INC, '/home/skipper/perl-lib';
};
```

Этот вариант дополнения списка @INC встречается чаще других, хотя Perl предоставляет специальную директиву use lib для этих целей, которая исполняется на этапе компиляции и поэтому полностью соответствует нашим ожиданиям. Она вставляет имя заданного каталога в начало списка @INC точно так же, как это происходило в предыдущем примере.

```
use lib qw(/home/skipper/perl-lib);
```

Однако путь к требуемому каталогу не всегда известен заранее. В предыдущих примерах мы жестко задавали имена каталогов. Но если мы собираемся устанавливать свои программы на другие машины, мы не можем быть заранее уверены в том, что имена каталогов с библиотеками будут в точности совпадать с нашими. Преодолеть это препятствие нам поможет модуль FindBin, поставляемый с дистрибутивом Perl. Он возвращает полный путь к каталогу со сценарием, посредством которого мы затем будем добавлять каталоги поиска в список.

```
use FindBin qw($Bin);
```

Теперь переменная `$Bin` будет содержать путь к каталогу с нашим сценарием. Если наши библиотеки находятся в этом же каталоге, то следующая строка программы имеет такой вид:

```
use lib $Bin;
```

Если библиотеки находятся во вложенных подкаталогах каталога со сценарием, то мы можем объединить компоненты пути к этим каталогам.

```
use lib "$Bin/lib"; # в подкаталоге

use lib "$Bin/../lib"; # на один уровень выше, а затем в каталог lib
```

Таким образом, если нам известны пути к каталогам с библиотеками, хотя бы относительно каталога со сценарием, то мы уже можем отказаться от жестко заданных имен каталогов. Такой подход делает сценарии более переносимыми.

Дополнение списка @INC с помощью переменной PERL5LIB

Шкипер хочет подключить библиотеки к своим программам, но для этого ему придется отредактировать все свои программы. Это дело может оказаться довольно трудоемким, но всего этого можно избежать, если записать список требуемых каталогов в переменную окружения `PERL5LIB`. Например, в C shell это делается так:

```
setenv PERL5LIB /home/skipper/perl-lib
```

А так в оболочке Борна и аналогичных:

```
PERL5LIB=/home/skipper/perl-lib; export PERL5LIB
```

Шкипер может сделать это один раз и забыть обо всех неприятностях. Однако если Джиллиган не выполнит те же самые настройки у себя, он не сможет пользоваться программами! Несмотря на все удобства, которые дает переменная окружения `PERL5LIB`, мы не можем полагаться на нее, если собираемся работать с одними и теми же программами совместно с другими пользователями. (Будет очень сложно заставить всех программистов добавить в свое окружение переменную `PERL5LIB`. Мы уже пробовали это сделать.)

Переменная `PERL5LIB` может содержать сразу несколько каталогов, для этого они должны отделяться друг от друга символом двоеточия. При запуске Perl отыщет эту переменную и добавит все каталоги, содержащиеся в ней, в список `@INC`.

Переменная `PERL5LIB` может быть создана системным администратором в одном из сценариев начальной загрузки системы, но такой подход не приветствуется. Назначение переменной `PERL5LIB` состоит в том, чтобы дать рядовым пользователям возможность расширить список каталогов поиска. Если системный администратор пожелает добавить какие-

либо каталоги в список по умолчанию, он может просто пересобрать и переустановить Perl с необходимыми настройками.

Дополнение списка @INC с помощью ключа -I

Если Джиллигану будет заранее известно, что в одной из программ Шкипера отсутствует указание на требуемый каталог, он сможет либо добавить полный путь к этому каталогу в переменную PERL5LIB, либо вызвать Perl с одним или более ключами -I. Например, команда, вызывающая программу Шкипера get_us_home, может выглядеть так:

```
perl -I/home/skipper/perl-lib /home/skipper/bin/get_us_home
```

Очевидно, что Джиллигану будет гораздо проще, если пути к дополнительным библиотекам определены в тексте самой программы. Но иногда без ключа -I не обойтись.[1] Этот прием эффективен, даже если программы Шкипера будут недоступны Джиллигану для записи. Он по-прежнему сможет читать эти файлы, например чтобы опробовать совместимость новой версии своей библиотеки с программами Шкипера.

Конфликт имен

Иногда Шкиперу не удается избежать столкновения корабля с островом, причем эти столкновения порой обусловлены простым совпадением имен подпрограмм. Предположим, что Шкипер сохранил все свои расчудесные и полезные подпрограммы в файле navigation.pm, а Джиллиган написал свой собственный навигационный пакет, куда включил подпрограмму head_toward_port:

```
#!/usr/bin/perl

require 'navigation.pm';

sub turn_toward_port {
    turn_toward_heading(compute_heading_to_island( ));
}

sub compute_heading_to_island {
    .. тело подпрограммы ..
}

.. остальной программный код ..
```

После этого Джиллиган отладил свою программу (возможно, не без помощи умнейшего человека, которого мы назовем Профессором), и у него все работало прекрасно.

---

[1] Операция расширения списка @INC с помощью переменной PERL5LIB или ключа командной строки -I автоматически добавляет в этот список все вложенные подкаталоги. Это упрощает установку дополнительных модулей, имена каталогов которых несут в себе информацию о номере версии или сведения о конкретной архитектуре, например объектные модули, написанные на языке C.

Однако теперь Шкипер решил добавить в библиотеку navigation.pm подпрограмму turn_toward_port, которая выполняет поворот влево на 45° (на жаргоне означает «курс домой»).

После этого программа Джиллигана будет терпеть постоянные неудачи, поскольку при попытке взять курс на порт она будет заставлять корабль кружиться на месте! Беда в том, что Perl скомпилирует сначала подпрограмму turn_toward_port Джиллигана, а затем уже на этапе исполнения выполнит директиву require и заместит ее подпрограммой turn_toward_port Шкипера. Конечно, если бы Джиллиган разрешил вывод предупреждений, он смог бы заметить эту ошибку, но почему он должен принимать это во внимание?

Дело в том, что под именем turn_toward_port Джиллиган подразумевал «turn toward the port on the island – взять курс на порт острова», тогда как Шкипер имел ввиду «turn toward the left – лево руля». Как разрешить это затруднение?

Один из способов заключается в том, чтобы Шкипер добавил префикс navigation_ к имени каждой подпрограммы в своей библиотеки. В этом случае программа Джиллигана приобрела бы такой вид:

```
#!/usr/bin/perl

require 'navigation.pm';

sub turn_toward_port {
    navigation_turn_toward_heading(compute_heading_to_island( ));
}

sub compute_heading_to_island {
    .. тело подпрограммы ..
}

.. остальной программный код ..
```

Совершенно очевидно, что функция navigation_turn_toward_heading находится в файле navigation.pm. Этот вариант прекрасно подходит Джиллигану, но очень неудобен для Шкипера, поскольку имена подпрограмм в его библиотеке значительно удлинились:

```
sub navigation_turn_toward_heading {
    .. тело подпрограммы ..
}

sub navigation_turn_toward_port {
    .. тело подпрограммы ..
}

1;
```

Теперь имя каждой скалярной переменной, каждого массива, каждого хеша и подпрограммы должно иметь префикс navigation_, чтобы гарантировать отсутствие потенциальных конфликтов имен с библиотеками других пользователей. Очевидно, что теперь плаванье на корабле превратится в муку для старого моряка. Что же делать?

Имена пакетов как разделители пространств имен

Все выглядело бы гораздо лучше, если бы в последнем примере можно было не добавлять префиксы. Обратимся к пакетам:

```
package Navigation;

sub turn_toward_heading {
    .. тело подпрограммы ..
}

sub turn_toward_port {
    .. тело подпрограммы ..
}

1;
```

Объявление `package` в начале файла сообщает Perl, что он должен подразумевать наличие префикса `Navigation::` перед большинством имен в файле. Таким образом, программный код, показанный выше, фактически означает:

```
sub Navigation::turn_toward_heading {
    .. тело подпрограммы ..
}

sub Navigation::turn_toward_port {
    .. тело подпрограммы ..
}

1;
```

Теперь, когда Джиллиган соберется воспользоваться этой библиотекой, он просто должен добавить префикс `Navigation::` к именам подпрограмм из этой библиотеки и опустить его, если обращается к одноименным подпрограммам из своей библиотеки:

```
#!/usr/bin/perl

require 'navigation.pm';

sub turn_toward_port {
    Navigation::turn_toward_heading(compute_heading_to_island( ));
}

sub compute_heading_to_island {
    .. тело подпрограммы ..
}
.. остальной программный код ..
```

Порядок именования пакетов подчиняется тем же правилам, что и имена переменных: они могут содержать алфавитно-цифровые символы и символ подчеркивания, но не могут начинаться с цифровых символов. Кроме того, по причинам, которые описаны в документе `perlmodlib`,

имена пакетов должны начинаться с заглавного символа и не повторять имена пакетов, включенных в CPAN или в базовый дистрибутив Perl. Названия пакетов могут также состоять из нескольких имен, разделенных двумя двоеточиями, например `Minnow::Navigation` или `Minnow::Food::Storage`.

Практически все имена скаляров, массивов, хешей, подпрограмм и дескрипторов файлов[1] предваряются именем пакета, если имя не содержит один или более маркеров в виде двойного двоеточия.

Таким образом, внутри пакета `navigation.pm` мы можем использовать следующие переменные:[2]

```
package Navigation;
@homeport = (21.283, -157.842);

sub turn_toward_port {
    .. программный код ..
}
```

Мы можем обратиться к переменной `@homeport` из основной программы, добавив к ее имени название пакета:

```
@destination = @Navigation::homeport;
```

Если считать, что все имена, встречающиеся в пакете, должны предваряться именем самого пакета, тогда что можно сказать по поводу имен в основной программе? Да, они тоже находятся в пакете с именем `main`, как если бы в начале файла стояла директива `package main`. Таким образом, чтобы уберечь Джиллигана от использования, скажем, `Navigation::turn_toward_heading`, в файл `navigation.pm` можно вставить такое определение:

```
sub main::turn_toward_heading {
    .. тело подпрограммы ..
}
```

Теперь подпрограмма будет считаться определенной в основной программе – в пакете `main`, а не в пакете `Navigation`. Это далеко не самое лучшее решение (его мы продемонстрируем в главе 15, когда будем говорить о модуле `Exporter`); во всяком случае в пакете `main` нет ничего особенного по сравнению с другими пакетами.

Практически то же самое мы делали в главе 3, когда импортировали имена в наши сценарии, правда, тогда мы не раскрывали всей подноготной. Тогда мы импортировали имена подпрограмм и переменных в текущий пакет (тот же самый пакет `main`). Благодаря этому в теку-

---

1 За исключением лексических переменных, в чем вы скоро убедитесь.

2 Маленькое примечание: в координатах 21,283 градуса северной широты и 157,842 градуса западной долготы находится вполне реальная пристань, где проходили съемки известного телевизионного сериала.

щем пакете были доступны только импортированные имена, если мы не указывали полное имя, включающее имя пакета. Более подробно об этом мы поговорим немного позже.

Область видимости директивы package

Во всех файлах по умолчанию предполагается наличие директивы `package main;`.[1] Область действия любой директивы `package` распространяется до следующей директивы `package`, если эта директива не находится в фигурных скобках; в противном случае Perl запоминает имя пакета и восстанавливает его, встретив закрывающую фигурную скобку. Например:

```
package Navigation;

{ # начало области видимости блока
    package main; # здесь область видимости пакета main

    sub turn_toward_heading { # main::turn_toward_heading
        .. тело подпрограммы ..
    }
} # конец области видимости блока

# возврат к области видимости пакета Navigation

sub turn_toward_port { # Navigation::turn_toward_port
    .. тело подпрограммы ..
}
```

Лексическая область видимости пакета определяется так же, как область видимости переменных, объявленных со спецификатором `my`. Она ограничивается либо областью видимости блока, либо границами файла, в котором объявлено имя пакета.

В большинстве библиотек имеется только одно объявление имени пакета – в самом начале файла. В большинстве программ имя пакета `main` не изменяется. Однако вы должны знать, что имя пакета можно временно изменить.[2]

---

1 Язык Perl не заставляет объявлять функцию `main()`, в отличие от C. Perl знает, что в каждом сценарии должно быть такое объявление, и потому подразумевает его наличие по умолчанию.

2 Некоторые имена всегда определены в пакете `main` независимо от текущего имени пакета: `ARGV`, `ARGVOUT`, `ENV`, `INC`, `SIG`, `STDERR`, `STDIN` и `STDOUT`. Мы всегда можем обратиться к имени `@INC`, пребывая в полной уверенности, что получаем доступ к `main::@INC`. Служебные переменные, такие как `$_`, `$@` и `$!`, либо все будут лексическими, либо принудительно будут помещены в пакет `main`. Таким образом, обращаясь к переменной `$.`, мы никогда не получим доступ к переменной `$Navigation::.` по ошибке.

Пакеты и лексические переменные

Лексические переменные (переменные, объявленные со спецификатором `my`) не предваряются именем текущего пакета, поскольку переменные пакета всегда *глобальные*: мы всегда можем обратиться к переменной пакета, если нам известно ее полное имя. Лексические переменные обычно временные и доступны только внутри ограниченной области программы. Если мы объявляем лексическую переменную и затем обращаемся к ней, не указывая имя пакета в качестве префикса, то получаем доступ к лексической переменной. Если же имя переменной предваряется именем пакета, то мы получаем доступ к глобальной переменной, а не к лексической.

Допустим, что одна из подпрограмм в пакете `navigation.pm` объявляет лексическую переменную `@homeport`. Тогда любое обращение к имени `@homeport` будет означать обращение к лексической переменной, а обращение по полному имени `@Navigation::homeport` с добавлением имени пакета будет означать обращение к переменной пакета.

```
package Navigation;
@homeport = (21.283, -157.842);

sub get_me_home {
    my @homeport;
    .. @homeport .. # обращение к лексической переменной
    .. @Navigation::homeport .. # обращение к переменной пакета
}

.. @homeport .. # обращение к переменной пакета
```

Вполне очевидно, что такая чехарда с именами усложняет понимание программного кода, поэтому желательно избегать такого дублирования имен. Однако результаты работы такого кода вполне предсказуемы.

Упражнения

Ответы на эти вопросы вы найдете в приложении А в разделе «Ответы к главе 10».

Упражнение 1 [25 мин]

Аборигены острова Угабугу придумали весьма необычные названия дней недели и месяцев. Ниже приводится простой, но не очень хорошо написанный Джиллиганом программный код. Исправьте его, добавьте функцию преобразования названий месяцев и оформите все это в виде библиотеки. В качестве дополнительного задания добавьте возможность проверки ошибок и подумайте о том, что можно было бы написать в документации к получившейся библиотеке.

```
@day = qw(арк дип уап сен поп сеп кир);
sub number_to_day_name { my $num = shift @_; $day[$num]; }
@month = qw(диз под бод род сип уакс лин сен кун физ нап деп);
```

Упражнение 2 [15 мин]

Напишите программу, которая с помощью вашей библиотеки (см. упражнение 1) и нижеследующего программного кода выводила бы сообщение (например, `Сегодня дип, сен 15, 2011`, означающее, что сегодня понедельник, 15 августа, 2011 года). Подсказка: функция `localtime` может возвращать номер дня недели, месяца и года совсем не так, как вы ожидаете, поэтому вы должны обратиться к соответствующей документации.

```
my($sec, $min, $hour, $mday, $mon, $year, $wday) = localtime;
```

11

Введение в объекты

Методология объектно-ориентированного программирования (ООП) помогает ускорить разработку программ и упрощает их сопровождение за счет организации программного кода в виде именованных сущностей. Чтобы перейти к работе с объектами, надо немного ближе ознакомиться с основополагающими моментами, но потраченное время окупится.

Преимущества ООП становятся очевидными, когда объем программы (включая все внешние библиотеки и модули) превышает N строк программного кода. К сожалению, никто так и не смог точно определить значение N, однако для Perl эту величину можно считать приблизительно равной 1000. Если вся программа умещается в пару сотен строк, то применение объектов вряд ли целесообразно.

Как и ссылки, объекты были добавлены в Perl тогда, когда в обращении уже находилось изрядное количество программ, написанных на Perl ниже 5 версии. По этой причине перед разработчиками встала задача не поломать существующий синтаксис языка при введении новых конструкций. Удивительно, но единственным синтаксическим дополнением, которое потребовалось сделать для обеспечения работы с объектами, стал *вызов метода*, о чем мы вскоре поговорим подробнее. Однако чтобы понять смысл этого дополнения, необходимо провести некоторые исследования, поэтому продолжим.

Архитектура объектов Perl целиком построена на понятиях пакетов, подпрограмм и ссылок. Поэтому, если вы начали чтение книги с этой главы, пожалуйста, вернитесь назад и начните чтение с начала. Готовы? Тогда вперед!

Если бы мы могли говорить на языке зверей...

Совершенно очевидно, что наши герои, потерпевшие кораблекрушение, не смогут выжить, питаясь одними кокосовыми орехами и анана-

сами. К счастью, незадолго до их появления на острове здесь потерпела кораблекрушение баржа с домашними животными, и наши герои получили возможность разводить на острове животных.

Послушаем голоса всех животных, какие находятся на острове:

```
sub Cow::speak {
    print "Корова: му-у-у-у!\n";
}
sub Horse::speak {
    print "Лошадь: иго-го!\n";
}
sub Sheep::speak {
    print "Овца: бе-е-е-е!\n";
}

Cow::speak;
Horse::speak;
Sheep::speak;
```

В результате получилось:

```
Корова: му-у-у-у!
Лошадь: иго-го!
Овца: бе-е-е-е!
```

Ничего сногсшибательного, подпрограммы довольно простые, но каждая из них находится в отдельном пакете и вызывается с указанием полного имени, включающего имя пакета. Теперь попробуем заполнить нашими животными пастбище:

```
sub Cow::speak {
    print "Корова: му-у-у-у!\n";
}
sub Horse::speak {
    print "Лошадь: иго-го!\n";
}
sub Sheep::speak {
    print "Овца: бе-е-е-е!\n";
}

my @pasture = qw(Cow Cow Horse Sheep Sheep);
foreach my $beast (@pasture) {
    &{$beast."::speak"}; # символическая ссылка на подпрограмму
}
```

Результат:

```
Корова: му-у-у-у!
Корова: му-у-у-у!
Лошадь: иго-го!
Овца: бе-е-е-е!
Овца: бе-е-е-е!
```

Вот это да! Но операция разыменования символической ссылки в теле цикла выглядит как-то не очень хорошо. Здесь мы полагались на режим `no strict 'refs'`, но для больших программ его рекомендовать нельзя.[1] Для чего он нам понадобился? Мы хотели вызвать подпрограммы, расположенные внутри пакетов, поскольку имя пакета неотделимо от имени подпрограммы.

Вызов метода с помощью оператора «стрелка»

Класс – это группа сущностей, обладающих одинаковыми признаками и поведением. Пока что будем считать, что запись `Class->method` описывает вызов подпрограммы `method` из пакета `Class`. Метод – это объектно-ориентированная разновидность подпрограммы, и с этого момента вместо термина «подпрограмма» мы будем употреблять термин «метод».[2] Это не совсем правильно, но мы продолжим наше движение вперед небольшими шагами. Попробуем воспользоваться новым способом вызова методов:

```
sub Cow::speak {
    print "Корова: му-у-у-у!\n";
}
sub Horse::speak {
    print "Лошадь: иго-го!\n";
}
sub Sheep::speak {
    print "Овца: бе-е-е-е!\n";
}

Cow->speak;
Horse->speak;
Sheep->speak;
```

В результате мы получим то же самое:

```
Корова: му-у-у-у!
Лошадь: иго-го!
Овца: бе-е-е-е!
```

По-прежнему ничего особенного. Строк столько же, все элементы программы постоянные, никаких переменных. Однако теперь части имен можно отделить друг от друга:

---

1 Все примеры в этой книге представляют собой вполне работоспособный программный код, однако некоторые из них нарушают правила, предписываемые директивой `use strict`. Сделано это с целью облегчить их понимание. В конце главы мы покажем, как сделать программный код совместимым с режимом `strict`.

2 В языке Perl нет никаких различий между подпрограммами и методами. Обе эти конструкции получают список входных аргументов в виде `@_`, и в обоих случаях мы должны стремиться к созданию безошибочного программного кода.

```
my $beast = 'Cow';
$beast->speak; # вызов метода Cow->speak
```

Вот! Теперь имя пакета отделено от имени подпрограммы, и мы можем использовать значение переменной в качестве имени пакета. На этот раз программный код будет работать даже в режиме `use strict 'refs'`.

Перепишем пример с пастбищем, взяв на вооружение оператор «->»:

```
sub Cow::speak {
    print "Корова: му-у-у-у!\n";
}
sub Horse::speak {
    print "Лошадь: иго-го!\n";
}
sub Sheep::speak {
    print "Овца: бе-е-е-е!\n";
}

my @pasture = qw(Cow Cow Horse Sheep Sheep);
foreach my $beast (@pasture) {
    $beast->speak;
}
```

Теперь все животные мычат, ржут, блеют, и нам удалось избавиться от символических ссылок на подпрограммы!

Но взгляните еще раз на содержимое методов `speak`, все они имеют одинаковую структуру: оператор `print` выводит строку, содержащую название животного и произносимый им звук. Один из основных принципов ООП заключается в минимизации повторяющихся участков программного кода. Если мы пишем некоторый код всего один раз, то экономим время. Кроме того, этот код нам придется проверять и отлаживать всего один раз, т. е. мы сэкономим еще больше времени.

Итак, нам известно, что метод можно вызывать посредством оператора `->`, и мы сможем найти более простое решение тех же самых задач.

Дополнительный параметр при вызове метода

Код:

```
Class->method(@args)
```

пытается вызвать подпрограмму `Class::method` как:

```
Class::method('Class', @args);
```

(Если в этом случае метод не будет найден, в дело вступает механизм наследования, но об этом чуть позже.) Это означает, что в виде первого аргумента (возможно, единственного, если никаких аргументов подпрограмме не передается) метод получает имя класса. Поэтому метод `speak` класса `Sheep` можно переписать:

```
sub Sheep::speak {
    my $class = shift;
    print "$class: бе-е-е-е!\n";
}
```

Методы остальных животных могут быть переписаны аналогично:

```
sub Cow::speak {
    my $class = shift;
    print "$class: му-у-у-у!\n";
}
sub Horse::speak {
    my $class = shift;
    print "$class: иго-го!\n";
}
```

В любом случае в переменную `$class` попадает значение, соответствующее конкретному методу. Но в конечном счете мы получили ту же самую организацию программного кода. Можно ли вынести общие участки программного кода за пределы методов классов? Да, для этого надо вызвать другой метод в том же самом классе.

Вызов второго метода с целью упрощения

Мы можем из метода `speak` вызвать вспомогательный метод `sound`. Этот метод будет возвращать текст, обозначающий звук, издаваемый животным:

```
{ package Cow;
    sub sound { 'му-у-у-у' }
    sub speak {
        my $class = shift;
        print "$class: ", $class->sound, "!\n";
    }
}
```

Вызывая метод `Cow->speak`, мы получаем в переменной `$class` внутри метода `speak` название животного `Cow`. Этот метод в свою очередь вызывает метод `Cow->sound`, который возвращает строку `му-у-у-у`. Насколько сильно такое описание класса `Cow` будет отличаться от описания класса `Horse`?

```
{ package Horse;
    sub sound { 'иго-го' }
    sub speak {
        my $class = shift;
        print "$class: ", $class->sound, "!\n";
    }
}
```

Изменилось только имя пакета и звук, произносимый животным. Возникает другой вопрос: можно ли каким-нибудь образом использовать единый метод `speak` во всех классах животных? Да, через механизм наследования!

Определим пакет с именем Animal, который будет содержать общий метод speak:

```
{ package Animal;
    sub speak {
        my $class = shift;
        print "$class ", $class->sound, "!\n";
    }
}
```

Можно сказать, что каждое животное обладает чертами, общими для всех животных (класс Animal), и каждое из них издает свой звук:

```
{ package Cow;
    @ISA = qw(Animal);
    sub sound { "му-у-у-у" }
}
```

Обратите внимание: здесь появился массив @ISA. Мы вернемся к нему буквально через минуту.

Что теперь произойдет, когда мы попытаемся вызвать метод Cow->speak?

Сначала Perl создаст список входных аргументов метода. В данном случае он будет содержать единственную строку Cow. Затем Perl попытается отыскать метод Cow::speak. Такого метода в классе нет, и тогда Perl проверит наличие массива @Cow::ISA, в котором указывается имя родительского класса. Такой массив в нашем случае имеется, и он содержит единственное имя Animal.

Теперь Perl попытается отыскать метод speak в классе Animal, то есть Animal::speak. Такой метод у нас есть, и поэтому Perl вызовет его и передаст ему список аргументов, созданный ранее:

```
Animal::speak('Cow');
```

Внутри метода Animal::speak первый аргумент, строка Cow, будет перемещен в переменную $class. На следующем шаге в процессе работы оператора print будет вызван метод $class->sound, который превратится в вызов Cow->sound:

```
print "$class: ", $class->sound, "!\n";
# но в переменной $class находится строка Cow, поэтому...
print 'Cow: ', Cow->sound, "!\n";
# где будет вызван метод Cow->sound, который вернет строку 'му-у-у-у',
# отсюда конечный результат получается следующим
print 'Cow: ', 'му-у-у-у', "!\n";
```

В результате мы получим то, что должны были получить.

Несколько замечаний о массиве @ISA

Эта магическая переменная @ISA (произносится «иза» (и ни в коем случае не «айса»), от английского «is a» – представляет собой) объявляет,

что класс корова (Cow) «представляет собой» животное (Animal).[1] Обратите внимание: это именно массив, а не скалярная переменная, потому что в редких случаях есть смысл вести поиск отсутствующих методов в нескольких родительских классах. Но об этом мы поговорим позже.

Если бы в классе Animal также был массив @ISA, то Perl проверил бы и его.[2] Обычно каждый из массивов @ISA содержит лишь один элемент (наличие нескольких элементов означает множественное наследование и многократно усиленную головную боль), в результате мы получаем стройное дерево наследования.[3]

Включив режим use strict, мы получим массу предупреждений, касающихся @ISA, потому что имя этой переменной не содержит явного имени пакета, и при этом она не является лексической (my) переменной. Мы не можем объявить эту переменную лексической, потому что она должна принадлежать пакету, чтобы быть доступной для механизма наследования.

Есть пара достаточно прямолинейных способов объявления и начальной установки переменной @ISA. Самый простой состоит в том, чтобы добавить имя пакета к имени переменной:

```
@Cow::ISA = qw(Animal);
```

Мы можем объявить ее как неявную переменную пакета:

```
package Cow;
use vars qw(@ISA);
@ISA = qw(Animal);
```

Если версия Perl достаточно современная (5.6 или выше), то можно применить спецификатор our:

```
package Cow;
our @ISA = qw(Animal);
```

Но если программный код может использоваться людьми, у которых установлена версия Perl 5.005 или ниже, тогда лучше от него отказаться.

Если мы создаем класс в отдельном файле на основе существующего объектно-ориентированного модуля, то строки:

```
package Cow;
use Animal;
```

---

[1] Фактически имя ISA является лингвистическим термином. В который раз лингвистическое прошлое Ларри Уолла отразилось на языке Perl.

[2] Поиск производится рекурсивно вглубь и слева направо в каждом массиве @ISA.

[3] Существует возможность организации наследования через UNIVERSAL и AUTOLOAD. Более подробную информацию по этой теме вы найдете на страницах справочного руководства perlobj.

```
use vars qw(@ISA);
@ISA = qw(Animal);
```

можно заменить на:

```
package Cow;
use base qw(Animal);
```

Такая форма записи намного компактнее. Кроме того, директива `use base` дает дополнительное преимущество, которое заключается в том, что она исполняется на этапе компиляции, благодаря чему игнорируются потенциальные ошибки, связанные с установкой переменной `@ISA` на этапе исполнения.

Перекрытие методов

Давайте добавим мышь, которую нельзя увидеть, зато можно услышать:

```
{ package Animal;
    sub speak {
        my $class = shift;
        print "$class: ", $class->sound, "!\n";
    }
}

{ package Mouse;
    @ISA = qw(Animal);
    sub sound { 'пи-пи-пи' }
    sub speak {
        my $class = shift;
        print "$class: ", $class->sound, "!\n";
        print "[меня не видно, хотя и слышно!]\n";
    }
}

Mouse->speak;
```

В результате получится:

```
Mouse: пи-пи-пи!
[меня не видно, хотя и слышно!]
```

Здесь класс `Mouse` имеет собственный метод `speak`, поэтому обращение происходит к методу `Mouse->speak`, а не к `Animal->speak`. Этот прием известен как *перекрытие методов*. Перекрытие методов служит для сокрытия методов родительского класса в дочернем классе (`Mouse`), когда в дочернем классе необходимо определить специализированную версию метода, который должен вызываться вместо метода родительского класса (`Animal`). Фактически теперь нам даже не нужно объявлять и инициализировать переменную `@Mouse::ISA`, поскольку класс `Mouse` обладает полным комплектом методов.

Теперь у нас появился программный код, до некоторой степени повторяющий содержимое метода Animal->speak, что может существенно осложнить его сопровождение. Предположим, что некто решил видоизменить текст, выводимый методом класса Animal, и добавить в него слово «кричит». Тогда он идет в класс Animal и вносит необходимые изменения. Но при обращении к классу Mouse эти изменения окажутся незаметными. Дело в том, что мы просто скопировали метод speak из класса Animal в класс Mouse, однако в ООП такой подход считается нарушением. Мы должны взять существующий программный код, а не создавать его копии.

Можно ли избежать этого? Можно ли как-нибудь определить, что мышь делает все то же, что и другие животные, но добавить примечание? Конечно!

В первую очередь попробуем обратиться к методу Animal::speak напрямую:

```
{ package Animal;
    sub speak {
        my $class = shift;
        print "$class: ", $class->sound, "!\n";
    }
}

{ package Mouse;
    @ISA = qw(Animal);
    sub sound { 'пи-пи-пи' }
    sub speak {
        my $class = shift;
        Animal::speak($class);
        print "[меня не видно, хотя и слышно!]\n";
    }
}
```

Обратите внимание: здесь не используется оператор ->, поэтому необходимо включить значение переменной $class в список аргументов (конечно, это строка Mouse) в качестве первого параметра функции Animal::speak.

А почему мы здесь обошлись без оператора «стрелка»? Представьте себе, что мы пытаемся вызвать метод Animal->speak, тогда первым аргументом ему будет передана строка "Animal", а не "Mouse", и когда придет время обратиться к методу sound, у нас не будет допустимого имени класса, чтобы выбрать корректный метод.

Однако прямой вызов метода Animal::speak может привести к появлению ошибок. Что если метод Animal::speak вообще не существует, а унаследован от другого класса, упомянутого в списке @Animal::ISA? Предположим, что иерархия классов имеет следующий вид:

```
{ package LivingCreature;
  sub speak { ... }
```

```
      ...
    }

{ package Animal;
    @ISA = qw(LivingCreature);
    # определение метода speak() отсутствует
    ...
}

{ package Mouse;
    @ISA = qw(Animal);
    sub speak {
        ...
        Animal::speak( ... );
    }
    ...
}
```

Мы отказались то оператора «стрелка», и у нас остается единственный шанс обратиться к нужному методу, потому что мы вызвали метод как обычную подпрограмму без привлечения механизма наследования. Perl попытается отыскать эту подпрограмму в пакете `Animal`, не найдет ее и прекратит исполнение программы.

Теперь выбор метода жестко зависит от имени класса `Animal`. Такой подход осложнит сопровождение программного кода, если кто-то вдруг изменит содержимое списка `@ISA` в классе `Mouse` и не обратит внимания на имя класса `Animal` в методе `speak`. Следовательно, это не совсем правильный путь.

Поиск унаследованного метода

Лучшее решение заключается в том, чтобы сообщить Perl о необходимости отыскать желаемый метод в цепочке наследования:

```
{ package Animal;
    sub speak {
        my $class = shift;
        print "$class: ", $class->sound, "!\n";
    }
}

{ package Mouse;
    @ISA = qw(Animal);
    sub sound { 'пи-пи-пи' }
    sub speak {
        my $class = shift;
        $class->Animal::speak(@_);
        print "[меня не видно, хотя и слышно!]\n";
    }
}
```

Выглядит немногим лучше, зато работает. Благодаря такому синтаксису поиск метода speak начинается с класса Animal и продолжается дальше по цепочке наследования, если в этом классе искомый метод не будет обнаружен. Первым параметром методу будет передано содержимое переменной $class (поскольку мы опять задействовали оператор «стрелка»), таким образом, найденный метод speak получит имя класса Mouse и сможет вызвать метод Mouse::sound.

Однако это далеко не самое лучшее решение. Мы по-прежнему вынуждены помнить о содержимом @ISA и соответственно определять начальный класс поиска метода (содержимое одного должно точно соответствовать другому). Было бы еще хуже, если бы класс Mouse имел несколько элементов в массиве @ISA, а мы при этом не знали бы, в каком из родительских классов определен метод speak.

А есть ли способ еще лучше?

SUPER способ добиться того же самого

Заменив в вызове метода имя класса Animal на служебное слово SUPER, мы сможем потребовать от Perl автоматически выполнить поиск требуемого метода во всех суперклассах (классы, перечисленные в списке @ISA):

```
{ package Animal;
    sub speak {
        my $class = shift;
        print "$class: ", $class->sound, "!\n";
    }
}

{ package Mouse;
    @ISA = qw(Animal);
    sub sound { 'пи-пи-пи' }
    sub speak {
        my $class = shift;
        $class->SUPER::speak;
        print "[меня не видно, хотя и слышно!]\n";
    }
}
```

Таким образом, обращение SUPER::speak говорит о том, что поиск метода speak должен выполняться в классах, которые перечислены в списке @ISA текущего пакета, после чего должен быть вызван первый из найденных методов. В данном случае будет найден только один класс Animal, в этом классе будет обнаружен метод speak, после чего этот метод будет вызван, а в качестве первого аргумента ему будет передано имя класса Mouse.

Зачем нужен аргумент @_

В последнем примере методу speak могли бы передаваться дополнительные параметры (например, параметры, определяющие, сколько раз должно прокричать животное или высоту голоса), которые игнорируются методом Mouse::speak. Передать их методу родительского класса можно, добавив их в вызов этого метода:

```
$class->SUPER::speak(@_);
```

В данной строке вызывается метод speak родительского класса, которому передаются все входные аргументы, еще не вытолкнутые из текущего списка входных аргументов.

Как правильнее сделать? Это зависит от предъявляемых требований. Если мы реализуем класс, который лишь добавляет некоторые индивидуальные черты к поведению родительского класса, лучше просто передать все аргументы, которые оказались не востребованными текущим методом. Но если нам необходимо точно определить поведение родительского класса, мы должны явно передавать точные значения аргументов.

Что мы узнали…

К настоящему моменту мы изучили синтаксис обращения к методу с помощью оператора «стрелка»:

```
Class->method(@args);
```

или эквивалентную форму записи:

```
my $beast = 'Class';
$beast->method(@args);
```

в которой сначала создается список аргументов:

```
('Class', @args)
```

а затем предпринимается попытка вызвать метод:

```
Class::method('Class', @args);
```

Если Perl не сможет найти Class::method, он обойдет элементы массива @Class::ISA (рекурсивно) и попытается отыскать пакет, который содержит метод method, после чего вызовет его.

В главе 12 будет показано, как различать отдельных животных друг от друга, назначая им свойства, называемые *переменными экземпляров*.

Упражнения

Ответы на эти вопросы вы найдете в приложении A в разделе «Ответы к главе 11».

Упражнение 1 [20 мин]

Напишите определения классов `Cow`, `Horse`, `Sheep` и `Mouse` так, чтобы они могли работать в режиме `use strict`. Для этого, если вы работаете с современной версией Perl, воспользуйтесь спецификатором `our`. Ваша программа должна запрашивать у пользователя название одного или более животных, создавать пастбище с этими животными, а затем заставлять каждое из них подать голос.

Упражнение 2 [40 мин]

Добавьте в программу класс `Person` на том же уровне, что и класс `Animal`, так, чтобы оба эти класса наследовали класс `LivingCreature`. Добавьте к методу `speak` класса `Person` дополнительный параметр, в котором можно было бы передавать произносимую фразу. При отсутствии этого параметра метод (класса `Person`) должен воспроизводить какой-либо звук (например, «ля-ля-ля»). Поскольку это не доктор Дулитл, вы должны реализовать методы `speak` таким образом, чтобы животные не могли говорить. (То есть метод `speak` класса `Animal` должен игнорировать дополнительные параметры.) Попытайтесь избежать дублирования программного кода и не допустить появления ошибок, например в случае, если вы забыли определить метод `sound` для какого-нибудь животного.

Продемонстрируйте работу класса `Person`, обратившись к методу этого класса без дополнительного параметра, а затем повторно вызвав его с какой-либо фразой в качестве параметра.

12

Объекты и данные

Простой синтаксис, рассмотренный в главе 11, позволяет создавать методы классов, задействовать механизм наследования (множественного), перекрывать унаследованные методы и расширять функциональность классов. Мы научились описывать общий программный код в одном месте и разобрались с возможностью многократного использования одной и той же реализации с некоторыми изменениями и дополнениями. Все это составляет основу объектов, но каждый экземпляр того или иного объекта может иметь собственные данные, которые называются *данными экземпляра*, о чем мы с вами еще не говорили.

Лошадь лошади рознь

Вернемся к программному коду из главы 11, который описывает классы `Animal` и `Horse`:

```
{ package Animal;
    sub speak {
        my $class = shift;
        print "$class: ", $class->sound, "!\n"
    }
}

{ package Horse;
    @ISA = qw(Animal);
    sub sound { 'иго-го' }
}
```

Такое определение классов дает нам возможность вызвать метод `Horse->speak`, что приводит к вызову метода класса `Animal::speak`, который в свою очередь вызывает метод `Horse::sound`, чтобы получить звук, издаваемый лошадью:

```
Horse: иго-го!
```

Но в такой реализации все объекты класса Horse будут абсолютно идентичны друг другу. Если мы добавим какой-либо метод, он автоматически будет совместно использоваться всеми лошадьми. Это отменный способ сделать всех лошадей одинаковыми, но как отличить их друг от друга? Предположим, что мы дали каждой лошади кличку. Должен же быть способ, позволяющий хранить клички отдельно друг от друга.

Этого можно добиться, если создать экземпляры. Экземпляры создаются классом, точно так же как автомобили создаются на автомобильной фабрике. Каждый экземпляр может иметь свойства, которые называются *переменными экземпляра* (или переменными-членами, если следовать терминологии C++ или Java). Экземпляры имеют уникальный идентификатор (как, например, регистрационный номер лошади), общие свойства (цвет или порода лошади) и общие черты поведения (например, при натягивании узды лошадь останавливается).

В языке Perl каждый экземпляр должен представлять собой ссылку на один из встроенных типов. Простейшая разновидность ссылки, в которой можно сохранить кличку лошади, – это ссылка на скаляр:[1]

```
my $name = 'мистер Эд'; 2
my $tv_horse = \$name;
```

Теперь переменная $tv_horse представляет собой ссылку на данные (кличку), которые будут относиться к конкретному экземпляру. Заключительный шаг превращения ссылки в реальный экземпляр состоит в обращении к оператору bless:

```
bless $tv_horse, 'Horse';
```

Оператор bless следует по ссылке и отыскивает переменную, на которую она указывает, в данном случае это скалярная переменная $name. Затем он «освящает» эту переменную при помощи оператора bless (to bless – освящать), превращая ее в объект класса Horse. (Представьте себе, что к переменной $name приклеен листочек с надписью Horse.)

С этого момента $tv_horse становится экземпляром класса Horse.[3] Таким образом, мы получаем вполне конкретный экземпляр лошади. Сама ссылка во всех отношениях остается обычной ссылкой и может быть разыменована обычным способом.[4]

---

1 Это самый простой, но по причинам, о которых мы вскоре расскажем, редко применяемый способ.

2 Мистер Эд – говорящая лошадь из американского телесериала («Mr. Ed», 1961 г.)

3 Фактически $tv_horse указывает на объект, но ссылки на объекты практически повсеместно принято называть объектами. Следовательно, вполне допустимо говорить, что $tv_horse есть объект класса Horse, а не переменная, ссылающаяся на объект.

4 Хотя лучше не делать это вне контекста класса, а о причинах вы узнаете позже.

Вызов метода экземпляра

Как и в случае с именами пакетов (классов), метод экземпляра можно вызвать посредством оператора «стрелка». Послушаем, какой звук издает `$tv_horse`:

```
my $noise = $tv_horse->sound;
```

Прежде чем вызвать метод `sound`, Perl сначала замечает, что `$tv_horse` представляет собой ссылку на объект, то есть экземпляр класса. Затем он создает список параметров – точно так же, как это происходило, когда мы вызывали метод класса с помощью оператора `->`. В данном случае это будет единственный параметр `($tv_horse)`. (Позже мы покажем, что остальные аргументы занимают свои места в списке вслед за переменной экземпляра – точно так же, как и в случае обращения к методам класса.)

Затем происходит самое интересное: Perl берет имя класса, экземпляром которого является переменная – в данном случае это будет класс `Horse` – и с его помощью отыскивает и вызывает требуемый метод, как если бы мы вместо `$tv_horse->sound` записали `Horse->sound`. В этом и состоит суть оператора `bless` – связать имя класса со ссылкой, чтобы иметь возможность отыскать нужный метод.

В данном случае Perl отыскивает метод `Horse::sound` (не прибегая к механизму наследования) и выполняет вызов подпрограммы:

```
Horse::sound($tv_horse)
```

Обратите внимание: в качестве первого аргумента передается ссылка на экземпляр, а не имя класса, как прежде. Метод возвращает строку «иго-го», которая затем записывается в переменную `$noise`.

Если бы Perl не смог отыскать подпрограмму `Horse::sound`, он обратился бы к массиву `@Horse::ISA` и попытался отыскать требуемый метод в одном из суперклассов, как и в случае вызова метода по имени класса. Единственное отличие между вызовом метода по имени класса и вызовом метода экземпляра заключается в том, что будет передано в первом аргументе – экземпляр класса (связанная ссылка) или имя класса (строка).[1]

Доступ к данным экземпляра

В первом аргументе мы получаем экземпляр класса, поэтому у нас есть возможность извлечь данные, характерные именно для этого экземпляра. Добавим в класс возможность получить кличку лошади:

```
{ package Horse;
    @ISA = qw(Animal);
```

[1] Вероятно, такая техника отличается от принятой в других объектно-ориентированных языках.

```
        sub sound { 'иго-го' }
        sub name {
            my $self = shift;
            $$self;
        }
    }
```

А теперь прочитаем ее:

```
    print $tv_horse->name, ": ", $tv_horse->sound, "\n";
```

Внутри метода Horse::name массив @_ будет содержать переменную $tv_horse, которая в самом начале метода выталкивается из массива и записывается в переменную $self. Уже стало традицией в методах экземпляров перемещать первый аргумент в переменную экземпляра класса с именем $self. Желательно, чтобы вы также старались придерживаться этой традиции, если у вас нет достаточно веских оснований, чтобы не следовать ей (в языке Perl имя $self не имеет какого-то особого значения).[1] Затем мы разыменовываем $self как ссылку на скаляр и в результате получаем имя «мистер Эд»:

```
    мистер Эд: иго-го
```

Как создать лошадь

Если бы всех своих лошадей мы создавали вручную, как показано выше, то время от времени наверняка допускали бы ошибки. Кроме того, «вскрытие» каждой конкретной лошади нарушает один из принципов ООП. Такой способ нормален для ветеринара, но не для хозяина лошади. Поэтому позволим классу Horse самому создавать новые экземпляры самого себя:

```
    { package Horse;
        @ISA = qw(Animal);
        sub sound { 'иго-го' }
        sub name {
            my $self = shift;
            $$self;
        }

        sub named {
            my $class = shift;
            my $name = shift;
            bless \$name, $class;
        }
    }
```

[1] Если у вас есть опыт работы с другими объектно-ориентированными языками, то вы можете выбрать для этой переменной имя $this или $me, но тем самым вы наверняка введете в заблуждение других знатоков Perl, которые потом будут изучать ваш программный код.

Теперь с помощью метода `named` мы можем создать экземпляр класса `Horse`:

```
my $tv_horse = Horse->named('мистер Эд');
```

Мы снова вернулись к вызову метода класса. Метод `Horse::named` принимает два аргумента: `"Horse"` и `"мистер Эд"`. Оператор `bless` не только связывает переменную `$name` с именем класса, но и возвращает ссылку на эту переменную, которая затем уже выступает в роли возвращаемого значения метода. Так создаются лошади!

Здесь мы вызвали конструктор `named`, которому в качестве аргумента передали имя будущей лошади. Мы можем создать несколько конструкторов с разными именами вызова, чтобы разными способами «давать жизнь» объектам (например, регистрировать родословную или записывать дату рождения). Однако в большинстве случаев в классах создается единственный конструктор, как правило с именем `new`, который в состоянии порождать объекты разными способами в зависимости от значений своих аргументов. Любой из этих способов имеет право на существование при условии, что мы подробно документируем принципы использования конструкторов. В большинстве базовых модулей и модулей CPAN конструкторы имеют имя `new`, с известными исключениями, как, например, конструктор `DBI->connect()` в модуле `DBI`. Так или иначе, но выбор имени конструктора целиком зависит от автора класса.

Наследование конструктора

Есть ли что-нибудь в нашем конструкторе такое, что было бы специфично только для класса `Horse`? Нет. Таким образом, мы можем переместить конструктор в класс `Animal` и наследовать его во всех дочерних классах. Давайте попробуем:

```
{ package Animal;
    sub speak {
        my $class = shift;
        print "$class: ", $class->sound, "!\n"
    }
    sub name {
        my $self = shift;
        $$self;
    }
    sub named {
        my $class = shift;
        my $name = shift;
        bless \$name, $class;
    }
}

{ package Horse;
```

```
    @ISA = qw(Animal);
    sub sound { 'иго-го' }
}
```

Интересно, а что произойдет, если мы попробуем вызвать метод sound для экземпляра класса?

```
my $tv_horse = Horse->named('мистер Эд');
$tv_horse->speak;
```

Мы получим отладочное сообщение:

```
Horse=SCALAR(0xaca42ac): иго-го!
```

Но почему? Потому что метод Animal::speak ожидал получить в первом аргументе имя класса, а не переменную экземпляра. Когда вызывается метод экземпляра, ему передается связанная ссылка на скаляр со строкой, которая выводится так, как только что было показано, – как ссылка на строку, только с именем класса впереди.

Создание метода, работающего как с экземплярами, так и с классами

Нам необходим способ, которым можно было бы определить, в каком контексте был вызван метод – как метод класса или как метод экземпляра. Проще всего обратиться для этого к оператору ref. Он возвращает строку (с именем класса), когда ему передается связанная ссылка на экземпляр, и значение undef, когда он получает строку (например, имя класса). Для начала изменим метод name:

```
sub name {
    my $either = shift;
    ref $either
        ? $$either # это экземпляр класса - вернуть кличку
        : "$either без имени"; # это класс
}
```

Здесь с помощью оператора ?: определяется, что нужно вернуть – кличку конкретного экземпляра или строку с сообщением. Теперь метод name может вызываться и как метод экземпляра, и как метод класса. Обратите внимание: мы преднамеренно изменили имя переменной, в которой сохраняется значение первого аргумента, чтобы заострить ваше внимание:

```
print Horse->name, "\n"; # выведет "Horse без имени\n"

my $tv_horse = Horse->named('мистер Эд');
print $tv_horse->name, "\n"; # выведет "мистер Эд\n"
```

А теперь аналогичным образом исправим метод speak:

```
sub speak {
    my $either = shift;
```

```
        print $either->name, ': ', $either->sound, "\n";
    }
```

Поскольку метод sound уже отличает экземпляр класса от имени класса, работу по исправлению недостатков можно считать законченной!

Добавление параметров к методам

Попробуем научить наших животных есть:

```
{ package Animal;
    sub named {
        my $class = shift;
        my $name = shift;
        bless \$name, $class;
    }
    sub name {
        my $either = shift;
        ref $either
          ? $$either # экземпляр класса, вернуть кличку
          : "$either без имени"; # это класс
    }
    sub speak {
        my $either = shift;
        print $either->name, ': ', $either->sound, "\n";
    }
    sub eat {
        my $either = shift;
        my $food = shift;
        print $either->name, " ест $food.\n";
    }
}
{ package Horse;
    @ISA = qw(Animal);
    sub sound { 'иго-го' }
}
{ package Sheep;
    @ISA = qw(Animal);
    sub sound { 'бе-е-е-е' }
}
```

И проверим:

```
my $tv_horse = Horse->named('мистер Эд');
$tv_horse->eat('сено');
Sheep->eat('траву');
```

В результате получим:

```
мистер Эд ест сено.
Sheep без имени ест траву.
```

Методу экземпляра с параметрами при вызове передается ссылка на экземпляр и вслед за ним список параметров. Таким образом, первый вызов выглядит так:

```
Animal::eat($tv_horse, 'сено');
```

Методы экземпляров формируют *прикладной программный интерфейс* объекта (*Application Programming Interface – API*). Большая часть усилий, затрачиваемых на проектирование класса, связана с проработкой интерфейса, поскольку именно прикладной интерфейс определяет способы использования и сопровождения объекта и порожденных от него классов. Не торопитесь зафиксировать интерфейс, пока не разберетесь до конца, как вы (или другие) будете использовать объект.

Более сложные экземпляры

Как быть, если с экземпляром класса необходимо связать гораздо больший объем данных? Более сложные экземпляры классов могут содержать массу элементов, каждый из которых в свою очередь может быть ссылкой на другой объект. Пожалуй, проще всего хранить большие объемы сведений об экземплярах в хеше. Ключи хеша могут выступать в качестве названий частей объекта (они еще называются переменными экземпляра или переменными-членами) и содержать соответствующие значения, которые…, ну, в общем, значения.

Теперь надо решить, как превратить лошадь в фарш.[1] Вспомним, что объект – это ссылка, связанная с некоторой переменной. Поэтому мы запросто можем создать связанную ссылку на хеш и соответственно пересмотреть все наши взгляды на ссылку.

Попробуем создать овцу, которая будет иметь не только кличку, но и цвет:

```
my $lost = bless { Name => 'Бо', Color => 'белый' }, Sheep;
```

Кличка овцы `$lost->{Name}` – «Бо», а цвет `$lost->{Color}` – «белый». Но нам еще нужно, чтобы метод `$lost->name` возвращал кличку, что он сейчас сделать не сможет, потому что ожидает получить ссылку на скаляр. Исправить этот недостаток не составляет никакого труда:

```
## в классе Animal
sub name {
    my $either = shift;
    ref $either
      ? $either->{Name}
      : "$either без имени";
}
```

---

[1] Не обращаясь к мяснику. (Игра слов: hash (хеш) можно перевести с английского как «фарш». – *Примеч. перев.*)

Метод named по-прежнему создает скалярную овцу, исправим и его заодно:

```
## в классе Animal
sub named {
    my $class = shift;
    my $name = shift;
    my $self = { Name => $name, Color => $class->default_color };
    bless $self, $class;
}
```

Что это еще за default_color? Методу named может быть передана только кличка, и потому нам требуется как-то определить цвет, поэтому мы задаем цвет по умолчанию, специфичный для целого класса. Для овец мы выбрали белый цвет.

```
## в классе Sheep
sub default_color { 'белый' }
```

Теперь, чтобы избежать необходимости определять цвет по умолчанию для каждого класса, создадим в классе Animal метод, который будет возвращать цвет по умолчанию для всех животных:

```
## в классе Animal
sub default_color { 'коричневый' }
```

Таким образом, по умолчанию все животные у нас будут коричневого цвета (может быть, они грязные), если в конкретном классе не определен перекрывающий метод, возвращающий другой цвет по умолчанию.

Остальные методы мы оставим без изменений, поскольку лишь методы name и named зависели от структуры объекта. Таким способом обеспечивается соблюдение еще одного правила ООП: чем меньше внешний программный код полагается на внутреннюю структуру объекта, тем меньше исправлений придется в него вносить при изменении этой структуры.

Лошадь другого цвета

Будет довольно скучно, если все лошади у нас будут коричневого цвета. Добавим один-два метода, которые помогут нам узнавать и задавать цвет:

```
## в классе Animal
sub color {
    my $self = shift;
    $self->{Color};
}
sub set_color {
    my $self = shift;
    $self->{Color} = shift;
}
```

Теперь изменим цвет мистера Эда:

```
my $tv_horse = Horse->named('мистер Эд');
$tv_horse->set_color('черный с белыми пятнами');
print $tv_horse->name, ' окрашен в ', $tv_horse->color, " цвет\n";
```

В результате получается:

```
мистер Эд окрашен в черный с белыми пятнами цвет
```

Что возвращать

Исходя из стиля, в котором написан метод изменения значения свойства, он возвращает обновленное значение. При разработке методов изменения значений свойств обязательно нужно задумываться (и документировать) над тем, что он должен возвращать. На этот счет существуют разные мнения:

- Измененное значение параметра (то есть значение, которое было передано методу)
- Предыдущее значение (на манер функции `umask` или `select`, когда последняя вызывается с единственным аргументом)
- Сам объект
- Признак ошибки

Каждый из указанных вариантов имеет свои преимущества и недостатки. Например, если возвращается измененное значение, его можно использовать для изменения свойства другого объекта:

```
$tv_horse->set_color( $eating->set_color( color_from_user( ) ));
```

В реализации, приведенной выше, метод как раз возвращает обновленное значение. Зачастую это позволяет писать самый короткий программный код, который к тому же дает более высокую скорость работы.

Если возвращается предыдущее значение параметра, появляется возможность более простым способом реализовать временное изменение параметра:

```
{
    my $old_color = $tv_horse->set_color('оранжевый');
    ... выполнить действия с $tv_horse ...
    $tv_horse->set_color($old_color);
}
```

Такое поведение метода может быть реализовано следующим образом:

```
sub set_color {
    my $self = shift;
    my $old = $self->{Color};
    $self->{Color} = shift;
    $old;
}
```

Чтобы повысить эффективность метода, заставим с помощью функции wantarray метод не возвращать ничего, если вызывающий программный код игнорирует возвращаемое значение:

```
sub set_color {
    my $self = shift;
    if (defined wantarray) {
        # метод должен вернуть значение, поэтому
        # возвращаем его
        my $old = $self->{Color};
        $self->{Color} = shift;
        $old;
      } else {
        # вызывающий код игнорирует возвращаемое значение
        $self->{Color} = shift;
    }
}
```

Если возвращается сам объект, можно выполнить целую серию изменений:

```
my $tv_horse =
  Horse->named('мистер Эд')
       ->set_color('серый')
       ->set_age(4)
       ->set_height('17 пядей');1
```

Это стало возможным, потому что каждый из методов записи возвращает сам объект, который становится объектом для вызова следующего метода в цепочке. Реализация такого поведения метода тоже выглядит достаточно просто:

```
sub set_color {
    my $self = shift;
    $self->{Color} = shift;
    $self;
}
```

Здесь так же можно использовать трюк с возвращаемым значением, хотя в этом нет большого смысла, поскольку значение переменной $self уже было установлено.

Наконец, возврат признака ошибки может потребоваться в тех случаях, когда неудача операции изменения параметра не должна расцениваться как исключительное событие. В противном случае можно было бы возбудить исключение с помощью оператора die.

В общем вы можете реализовать любой из предложенных вариантов, но постарайтесь быть последовательными настолько, насколько это

---

[1] Здесь четыре последовательных вызова в первом операторе – часто порядок выполнения вызовов оказывается важным. – *Примеч. науч. ред.*

возможно, и обязательно документируйте свои решения (и не изменяйте их после того, как уже выпустили одну версию в свет).

Не открывайте черный ящик

Мы могли бы получить или изменить цвет вне контекста класса, обратившись к значению по ссылке на хеш: `tv_horse->{Color}`. Однако такой способ получения/изменения значений противоречит принципу *инкапсуляции* внутренней структуры объекта. Объект должен быть «черным ящиком», а мы «открываем крышку» и заглядываем внутрь.

Одна из целей ООП заключается в том, чтобы дать возможность разработчику класса `Animal` или `Horse` продолжать развивать реализацию внутренней структуры объекта и при этом гарантировать безукоризненную работу интерфейса. Чтобы понять, чем грозит прямой доступ к внутренней организации класса, предположим, что в классе `Animal` цвет хранится уже не в виде простого названия цвета, а в виде трех составляющих Красный-Зеленый-Синий (которые хранятся в виде ссылок на массив). В данном примере будет фигурировать вымышленный модуль `Color::Conversions`, с помощью которого будем изменять формат представления цвета:

```
use Color::Conversions qw(color_name_to_rgb rgb_to_color_name);
...
sub set_color {
    my $self = shift;
    my $new_color = shift;
    $self->{Color} = color_name_to_rgb($new_color); # ссылка на массив
}
sub color {
    my $self = shift;
    rgb_to_color_name($self->{Color}); # передается ссылка на массив
}
```

Мы по-прежнему сможем получать и узнавать цвет посредством старого интерфейса, если будем применять методы доступа, поскольку все необходимые преобразования происходят без нашего участия. Кроме того, мы сможем добавить новые методы для работы с составляющими цвета напрямую:

```
sub set_color_rgb {
    my $self = shift;
    $self->{Color} = [@_]; # составляющие цвета передаются в виде аргументов
}
sub get_color_rgb {
    my $self = shift;
    @{ $self->{Color} };   # вернуть список RGB
}
```

Если раньше имелся некоторый программный код, который напрямую получал или изменял цвет в `$tv_horse->{Color}`, теперь он уже не

сможет этого сделать. Он не сможет безнаказанно записать строку (`'голубой'`) туда, где нужна ссылка на массив (`[0,0, 255]`), или правильно интерпретировать ссылку на массив как строку. Поэтому в ООП рекомендуется создавать специальные методы доступа к элементам класса, даже если на это придется потратить некоторое время.[1]

Оптимизация методов доступа

Мы собираемся играть по правилам и получать доступ к свойствам объектов только с помощью специальных методов доступа, поэтому данные методы будут вызваться достаточно часто. Чтобы на каждом обращении к тому или иному методу сэкономить хоть немного времени, мы могли бы реализовать их так:

```
## в классе Animal
sub color { $_[0]->{Color} }
sub set_color { $_[0]->{Color} = $_[1] }
```

Мы не только уменьшили объем программного кода, но и немного его ускорили, хотя это едва ли можно будет уловить на общем фоне всего, что делается в программе. Здесь мы обращаемся к единственному элементу массива посредством `$_[0]`. Вместо того чтобы с помощью оператора `shift` перемещать аргумент в другую переменную, мы обращаемся к нему напрямую.

Операция чтения и записи в одном методе

В качестве альтернативы двум методам чтения и записи можно реализовать единственный метод, который будет совмещать в себе обе операции, различая их по присутствию дополнительных входных аргументов. Если дополнительных аргументов нет, значит, это метод чтения. А если есть, то метод записи. Простейшая версия такого метода:

```
sub color {
    my $self = shift;
    if (@_) { # есть дополнительные аргументы?
        # да, это метод записи:
        $self->{Color} = shift;
    } else {
        # нет, это метод чтения:
        $self->{Color};
    }
}
```

[1] В Perl «объектность» пришла в результате эволюции – в тех языках, которые изначально проектировались для поддержки ООП (C++, Java), вводятся квалификаторы области видимости (`public`, `protected`, `private`), которые делают невозможным доступ к членам экземпляра класса, не включенным в API класса. – *Примеч. науч. ред.*

Теперь мы можем написать такой программный код:

```
my $tv_horse = Horse->named('мистер Эд');
$tv_horse->color('черный с белыми пятнами');
print $tv_horse->name, ' окрашен в ', $tv_horse->color, " цвет\n";
```

Дополнительный аргумент во второй строке свидетельствует, что производится изменение цвета, а его отсутствие в третьей строке – о том, что выполняется чтение значения цвета.

Такая стратегия привлекает своей простотой, но у нее есть свои недостатки. Она замедляет скорость исполнения операции чтения, которая производится гораздо чаще, чем операция записи. Кроме того, такой подход осложняет поиск всех мест в программе, где производится изменение свойства, потому что приходится постоянно отфильтровывать операции чтения. Мы однажды обожглись на этой стратегии, когда неожиданно вместо операции чтения стала выполняться операция записи только потому, что другая функция (после изменений) стала возвращать иное число аргументов.

Ограничение доступа к методам только по имени класса или только для экземпляров класса

Операция записи клички лошади для всего класса Horse едва ли имеет хоть какой-то смысл. Аналогично никто не будет вызвать конструктор named для экземпляра класса. В Perl нельзя указать, что «этот метод является методом класса», а «этот – методом экземпляра», но благодаря оператору ref мы имеем возможность создать исключительную ситуацию, когда метод вызывается не по назначению. Например, в методе, который должен вызываться только как метод класса или только как метод экземпляра, мы можем проверить аргумент и определить, что делать дальше:

```
use Carp qw(croak);
sub instance_only {
    ref(my $self = shift) or croak "требуется ссылка на экземпляр класса";
    ... использовать $self как ссылку на экземпляр класса ...
}
sub class_only {
    ref(my $class = shift) and croak "требуется имя класса";
    ... использовать $class как имя класса ...
}
```

В случае экземпляра класса, который фактически представляет собой ссылку на связанную переменную, функция ref возвращает значение «истина» или значение «ложь», если ей будет передано имя класса, которое является обычной строкой. Когда функция ref возвращает нежелательное значение, мы вызываем функцию croak из модуля Carp (входит в состав стандартного дистрибутива Perl). Функция croak воз-

лагает ответственность на вызывающий программный код, создавая сообщение об ошибке, указывающее на точку вызова метода, а не на точку вызова самой функции croak. В результате мы получим сообщение об ошибке с номером строки, в которой был произведен ошибочный вызов метода, например:

```
требуется ссылка на экземпляр класса at their_code line 1234
```

Функция croak представляет собой альтернативную разновидность функции die. Кроме того, в модуле Carp имеется функция carp, которая может служить заменой функции warn. Обе они сообщают пользователю номер строки, из которой было произведено обращение к методу, вызвавшему ошибку. Если в своих модулях вместо die и warn вы задействуете функции модуля Carp, то ваши пользователи будут вам благодарны.

Упражнения

Ответы на эти вопросы вы найдете в приложении А в разделе «Ответы к главе 12» .

Упражнение [45 мин]

Добавьте в класс Animal возможность чтения и записи клички и цвета животного. Вы должны обеспечить работоспособность программного кода в режиме use strict. Кроме того, методы чтения должны иметь возможность работать как с экземпляром класса, так и с именем класса. Проверьте работоспособность с помощью следующего отрывка:

```
my $tv_horse = Horse->named('Эд');
$tv_horse->set_name('мистер Ed');
$tv_horse->set_color('серый');
print $tv_horse->name, ' окрашен в ', $tv_horse->color, " цвет\n";
print Sheep->name, ' окрашена в ', Sheep->color, ' цвет и издает звук ',
Sheep->sound, "\n";
```

Что следует предпринять, если будет произведена попытка изменить кличку или цвет для всего класса животных?

13

Уничтожение объектов

В двух предыдущих главах мы рассмотрели основные принципы создания объектов и взаимодействие с ними. Эта глава посвящена не менее важной теме – уничтожению объектов.

Как уже говорилось в главе 4, когда уничтожается последняя ссылка на структуру данных, Perl автоматически освобождает память, занимаемую этой структурой, включая уничтожение ссылок на другие структуры данных. Разумеется, это может привести к необходимости уничтожения других («вложенных») структур данных.

По умолчанию объекты работают аналогичным образом, потому что они используют все тот же механизм ссылок на структуры данных. Объект, являющийся ссылкой на хеш, уничтожается, когда уничтожается последняя ссылка на этот хеш. Если значениями элементов хеша являются ссылки, они удаляются обычным образом, что может привести к уничтожению других структур данных.

Уборка мусора

Предположим, что наш объект хранит во временном файле данные, которые не умещаются в памяти. Объект может включать в себя дескриптор этого временного файла в виде переменной экземпляра. В процессе уничтожения объекта дескриптор файла будет закрыт, но сам файл останется на диске, если не предпринять дополнительных действий по его уничтожению.

Чтобы выполнить заключительные операции по «уборке мусора» в процессе уничтожения объекта,[1] мы должны знать, когда это происходит. К счастью, Perl предоставляет возможность получить уведомление об

[1] Как первую фазу уничтожения объекта. – *Примеч. науч. ред.*

уничтожении объекта. Для этого достаточно реализовать в объекте метод деструктора DESTROY.

Когда уничтожается последняя ссылка на объект, скажем $bessie, Perl автоматически вызовет метод DESTROY для этого объекта, как если бы мы вставили строку:

```
$bessie->DESTROY
```

Такой вызов метода ничем не отличается от вызовов других методов: Perl начинает поиск метода с класса объекта и затем спускается по цепочке наследования, пока не обнаружит требуемый метод. Однако, в отличие от других, отсутствие этого метода не является ошибкой.[1]

Вернемся к нашему примеру класса Animal, который был определен в главе 11, и добавим в него исключительно в отладочных целях метод DESTROY, чтобы поймать момент, когда объект будет уничтожаться:

```
## в классе Animal
sub DESTROY {
    my $self = shift;
    print '[объект ', $self->name, " уничтожен.]\n";
}
```

Теперь, если в программе будут создаваться экземпляры каких-либо животных, мы будем получать извещения об их уничтожении. Например:

```
## добавить классы животных из предыдущей главы...

sub feed_a_cow_named {
    my $name = shift;
    my $cow = Cow->named($name);
    $cow->eat('траву');
    print "Возврат из подпрограммы.\n"; # в этой точке объект $cow
                                        # будет уничтожен
}
print "Начало программы.\n";
my $outer_cow = Cow->named('Бесси');
print "Появилась корова по кличке ", $outer_cow->name, ".\n";
feed_a_cow_named('Гвен');
print "После возврата из подпрограммы.\n";
```

В результате будет получено следующее:

```
Начало программы.
Появилась корова по кличке Бесси.
Гвен ест траву.
Возврат из подпрограммы.
[объект Гвен уничтожен.]
```

[1] Обращение к отсутствующим методам, как правило, приводит к появлению ошибки. Для того чтобы предотвратить это, достаточно вставить в базовый класс пустые заготовки методов.

```
После возврата из подпрограммы.
[объект Бесси уничтожен.]
```

Обратите внимание: корова по кличке «Гвен» создается внутри подпрограммы. Когда подпрограмма завершает свою работу, Perl замечает, что уничтожается последняя ссылка на объект `$cow`, и автоматически вызывает метод `DESTROY`, который выводит сообщение о том, что `объект Гвен уничтожен`.

А что же происходит в конце программы? Поскольку объекты не могут существовать за пределами программы, Perl делает завершающий обход всех оставшихся структур данных и уничтожает их. Это относится не только к лексическим переменным, но и к глобальным переменным пакета. Поскольку к концу программы «Бесси» еще жива, ее необходимо уничтожить, в результате чего мы получаем сообщение о «Бесси» в конце работы программы.[1]

Уничтожение вложенных объектов

Если объект содержит в себе другие объекты (например, в виде элементов массива или значений элементов хеша), Perl вызовет метод `DESTROY` объекта-контейнера еще до того, как начнется процесс уничтожения вложенных объектов. Это вполне разумно, поскольку объект-контейнер может использовать вложенные объекты для корректного завершения своей работы. Чтобы продемонстрировать это на конкретном примере, попробуем построить «коровник» и затем разрушить его. А чтобы сделать пример более интересным, создадим объект не в виде ссылки на хеш, а в виде ссылки на массив.

```
{ package Barn;
    sub new { bless [ ], shift }
    sub add { push @{+shift}, shift }
    sub contents { @{+shift} }
    sub DESTROY {
        my $self = shift;
        print "уничтожается объект $self...\n";
        for ($self->contents) {
            print '  ', $_->name, " стала бездомной.\n";
        }
    }
}
```

Здесь мы ограничились минимально необходимым определением объекта. Чтобы создать новый коровник, мы связываем ссылку на пустой массив с именем класса `Barn`, которое передается конструктору в виде

[1] Это происходит сразу же вслед за исполнением блоков `END` согласно тем же правилам, которым подчиняются блоки `END`: данный программный код исполняется в случае нормального, а не аварийного завершения программы.

первого аргумента. Добавление нового животного в коровник заключается в добавлении соответствующего объекта в конец массива. При запросе содержимого коровника, мы просто разыменовываем ссылку на массив объектов и возвращаем его содержимое.[1] Самая интересная часть в объекте – это деструктор. Он выводит сообщение об уничтожении самого объекта, а затем выводит клички животных, находившихся в коровнике. Следующая программа:

```
my $barn = Barn->new;
$barn->add(Cow->named('Бесси'));
$barn->add(Cow->named('Гвен'));
print "Коровник сгорел:\n";
$barn = undef;
print "Конец программы.\n";
```

выведет:

```
Коровник сгорел:
уничтожается объект Barn=ARRAY(0x541c)...
  Бесси стала бездомной.
  Гвен стала бездомной.
[объект Гвен уничтожен.]
[объект Бесси уничтожен.]
Конец программы.
```

Обратите внимание: Perl сначала разрушает коровник, позволяя нам получить клички животных, населявших его. Однако, как только коровник будет разрушен, вместе с ним исчезают все ссылки на животных, что вынуждает Perl вызвать деструкторы объектов животных. Сравните это со случаем, когда коровы живут за пределами коровника:

```
my $barn = Barn->new;
my @cows = (Cow->named('Бесси'), Cow->named('Гвен'));
$barn->add($_) for @cows;
print "Коровник сгорел:\n";
$barn = undef;
print "Пропали коровы:\n";
@cows = ( );
print "Конец программы.\n";
```

Эта программа выведет:

```
Коровник сгорел:
уничтожается объект Barn=ARRAY(0x541c)...
  Бесси стала бездомной.
```

[1] Задавались ли вы вопросом, зачем здесь стоит знак «плюс» (+)? Дело в том, что если бы мы просто указали `@{shift}`, то из-за того, что в фигурных скобках слово `shift` не имеет специальной приставки, оно интерпретировалось бы как символическая ссылка `@{"shift"}`. В языке Perl унарный плюс (символ плюса в начале строки) определен как пустая операция (даже не преобразует следующую за ним строку в число); именно таким способом можно отличить случаи, аналогичные данному.

```
  Гвен стала бездомной.
Пропали коровы:
[объект Гвен уничтожен.]
[объект Бесси уничтожен.]
Конец программы.
```

Теперь коровы продолжают жить до тех пор, пока не будут уничтожены другие ссылки на эти объекты (в массиве `@cows`).

Ссылки на коров будут уничтожены лишь после того, как деструктор коровника завершит свою работу. В случае необходимости у нас будет возможность выгнать коров из коровника, чтобы не дать им пропасть вместе со зданием. Для этого достаточно переместить объекты в другой массив.[1] Попробуем продемонстрировать это на примере другого класса `Barn2`:

```
{ package Barn2;
    sub new { bless [ ], shift }
    sub add { push @{+shift}, shift }
    sub contents { @{+shift} }
    sub DESTROY {
        my $self = shift;
        print "уничтожается объект $self...\n";
        while (@$self) {
            my $homeless = shift @$self;
            print ' ', $homeless->name, " стала бездомной.\n";
        }
    }
}
```

Теперь опробуем этот класс с тем же сценарием:

```
my $barn = Barn2->new;
$barn->add(Cow->named('Бесси'));
$barn->add(Cow->named('Гвен'));
print "Коровник сгорел:\n";
$barn = undef;
print "Конец программы.\n";
```

В результате получим:

```
Коровник сгорел:
уничтожается объект Barn2=ARRAY(0x541c)...
  Бесси стала бездомной.
[объект Бесси уничтожен.]
  Гвен стала бездомной.
[объект Гвен уничтожен.]
Конец программы.
```

---

[1] В случае использования хеша вместо массива для непосредственного удаления объектов из хеша можно обратиться к функции `delete`.

Как видите, Бесси лишилась дома сразу же, как только мы выгнали ее из коровника, а затем она была уничтожена. (Бедняжку Гвен постигла та же судьба.) Дело в том, что еще до окончания работы деструктора были уничтожены все ссылки на наших животных.

Теперь вернемся к нашей проблеме с временным файлом. Мы изменили класс Animal так, чтобы он создавал временный файл с помощью модуля File::Temp, который входит в состав стандартного дистрибутива Perl. Подпрограмма tempfile из этого модуля знает, как и где создавать временные файлы. Она возвращает дескриптор файла и его имя, а мы сохраняем оба эти значения, поскольку они нам потребуются в деструкторе.

```
## в классе Animal
use File::Temp qw(tempfile);

sub named {
    my $class = shift;
    my $name = shift;
    my $self = { Name => $name, Color => $class->default_color };
    ## начало нового программного кода...
    my ($fh, $filename) = tempfile( );
    $self->{temp_fh} = $fh;
    $self->{temp_filename} = $filename;
    ## конец нового программного кода...
    bless $self, $class;
}
```

Теперь у нас есть дескриптор файла и его имя, которые были сохранены в виде переменных экземпляра класса Animal (или любого другого, порожденного от класса Animal). В деструкторе мы закрываем файл и затем удаляем его[1]:

```
sub DESTROY {
    my $self = shift;
    my $fh = $self->{temp_fh};
    close $fh;
    unlink $self->{temp_filename};
    print '[объект ', $self->name, " уничтожен.]\n";
}
```

Когда будет уничтожена последняя ссылка на объект класса Animal (даже если это произойдет в результате завершения работы программы), автоматически будет удален и временный файл.

1 Эти же действия могут быть выполнены автоматически, но это лишило бы нас возможности проиллюстрировать наши рассуждения на примере. Обработка закрытия файла вручную позволит выполнить дополнительные действия, например переместить некоторые сведения из временного файла в базу данных.

Вторичная переработка

Деструкторы, как и любые другие методы, наследуются дочерними классами и могут вызываться с помощью механизма наследования, но точно так же существует возможность перекрытия деструкторов в дочерних классах. Например, мы можем решить, что павшие лошади могут быть как-то использованы. Для этого в нашем классе Horse мы перекроем метод DESTROY, унаследованный от класса Animal, и предусмотрим дополнительную переработку. Поскольку нам могут быть неизвестны все тонкости работы деструктора класса Animal, мы будем вызывать его посредством псевдокласса SUPER::, о котором рассказывалось в главе 11.

```
## в классе Horse
sub DESTROY {
    my $self = shift;
    $self->SUPER::DESTROY;
    print "[", $self->name, " отправлен на фабрику для переработки.]\n";
}

my @tv_horses = map Horse->named($_), ('Триггер', 'мистер Эд');
$_->eat('яблоко') for @tv_horses; # их последний прием пищи
print "Конец программы.\n";
```

Эта программа выведет:

```
Триггер ест яблоко.
мистер Эд ест яблоко.
Конец программы.
[объект мистер Эд уничтожен.]
[мистер Эд отправлен на фабрику для переработки.]
[объект Триггер уничтожен.]
[Триггер отправлен на фабрику для переработки.]
```

Напоследок мы накормили каждую из лошадей, а в конце программы был вызван деструктор каждой из них.

Первым делом новый деструктор вызывает деструктор родительского класса. Почему это так важно? Если унаследованный деструктор не будет вызван, то не будут выполнены действия, предусмотренные суперклассом этого класса. В принципе в этом нет ничего страшного, если родительский деструктор просто выводит отладочное сообщение, но если он, к примеру, удаляет временный файл, тогда окажется, что вы не удалили его!

В общем виде данное правило можно сформулировать так:

Всегда включайте в деструкторы вызов $self->SUPER::DESTROY *(даже если текущий класс не имеет базового/родительского класса).*

Точка вызова унаследованного деструктора (в конце или в начале) является предметом ожесточенных дискуссий. Если дочернему классу будут необходимы переменные экземпляра, унаследованные от супер-

класса, тогда унаследованный деструктор следует вызывать после выполнения всех необходимых действий, потому что деструктор родительского класса может изменять[1] их в процессе своей работы. С другой стороны, в нашем примере мы вызвали деструктор суперкласса в самом начале, потому что нам необходимо было сначала получить реакцию суперкласса.

Форма косвенного обращения к объектам

Синтаксис вызова метода с помощью оператора «стрелка» иногда называют синтаксисом *прямого обращения* к объекту, но кроме него существует синтаксис *косвенного обращения* к объекту, который не всегда работает правильно по причинам, о которых мы расскажем чуть ниже. Обращение к методу объекта с помощью оператора «стрелка»:

```
Class->class_method(@args);
$instance->instance_method(@other);
```

можно заменить формой записи, когда имя метода предшествует имени класса с последующим списком аргументов.

```
classmethod Class @args;
instancemethod $instance @other;
```

Такая форма записи значительно чаще встречалась в далекие дни Perl 5, и мы продолжаем пытаться искоренить ее. Очень жаль, что мы не можем рассказать о нем подробнее (если вы ничего не знаете об этом синтаксисе, то и не связывайтесь с ним), но он периодически будет встречаться вам, поэтому знать о нем совершенно необходимо. Такой синтаксис обычно встречается при вызове метода `new`, когда авторы модулей подменяют синтаксис, основанный на операторе «стрелка»:

```
my $obj = Some::Class->new(@constructor_params);
```

на то, что больше напоминает фразу, построенную в соответствии с правилами английского языка:

```
my $obj = new Some::Class @constructor_params;
```

и ближе для понимания программистам, знакомым с языком программирования C++. Разумеется, Perl не вкладывает никакого особого смысла в имя `new`, но сам синтаксис вызова кажется очень знакомым.

В чем же, собственно, заключаются недостатки, которые не позволяют подменять синтаксис на основе оператора > синтаксисом косвенного обращения к объекту? Представьте себе случай, когда экземпляр клас-

[1] Или, что гораздо хуже, просто уничтожить их. Вообще-то, по правилам классических языков ООП (C++, Java) деструктор суперкласса **неявно** вызывается после завершения деструктора производного класса, и этот порядок никаким способом нельзя изменить. – *Примеч. науч. ред.*

са представляет собой более сложную структуру, чем обычная скалярная переменная:

```
$somehash->{$somekey}->[42]->instance_method(@parms);
```

в такой ситуации мы не можем просто поменять местами обращения к методам, как это принято в синтаксисе косвенного обращения:

```
instance_method $somehash->{$somekey}->[42] @parms;
```

В косвенном синтаксисе допускаются лишь простые имена, не имеющие специального префикса (например, имена классов), простые скалярные переменные или фигурные скобки, обозначающие блок, который должен возвращать либо ссылку на экземпляр класса, либо имя класса.[1] Это означает, что в косвенном синтаксисе вызов метода оформляется примерно так:

```
instance_method { $somehash->{$somekey}->[42] } @parms;
```

Как быстро простое превратилось в уродливое. Есть и еще один недостаток: неоднозначность. Прорабатывая учебные примеры к теме косвенного синтаксиса обращения по ссылкам, мы написали:

```
my $cow = Cow->named('Бесси');
print name $cow, " ест.\n";
```

поскольку полагали, что такая форма записи эквивалентна следующей:

```
my $cow = Cow->named('Бесси');
print $cow->name, " ест.\n";
```

Однако, как потом выяснилось, этот пример ничего не выводил на экран. Тогда мы включили режим вывода предупреждений (ключ `-w` командной строки)[2] и получили такую последовательность предупреждений:

```
Unquoted string "name" may clash with future reserved word at ./foo line 92.
Name "main::name" used only once: possible typo at ./foo line 92.
print( ) on unopened filehandle name at ./foo line 92.
```

Данная строка была интерпретирована как:

```
print name ($cow, " ест.\n");
```

Другими словами, была выполнена операция вывода списка в файловый дескриптор с именем `name`. Конечно, мы хотели получить не это,

---

1 Самые проницательные читатели наверняка заметили, что те же самые правила применяются в косвенном синтаксисе обращения к дескрипторам файлов, равно как и правила, регулирующие работу со ссылками, подлежащими разыменованию.

2 Когда что-то происходит не так, как ожидалось, в первую очередь следует воспользоваться ключом `-w`. Возможно, это напоминание покажется вам излишним, поскольку при разработке программного кода включение режима вывода предупреждений – обычное дело.

поэтому должны были видоизменить обращение к методу, чтобы устранить неоднозначность.[1]

Таким образом, мы подошли к еще одному строгому правилу: *Всегда придерживайтесь синтаксиса прямого обращения к объектам*.

Тем не менее мы понимаем, что большинство программистов предпочитают писать `new Class ...`, а не `Class->new(...)`. В документации к старым модулям примеры следуют косвенному синтаксису обращения к классам, а попробовав один раз, вы подсознательно будете стремиться применять тот же прием везде. Однако иногда обстоятельства складываются так, что возникает неопределенность интерпретации программного кода (например, когда подпрограмма с именем `new` вызывается для класса, имя которого не ассоциируется с именем пакета). Когда у вас возникают сомнения в однозначности интерпретации косвенного синтаксиса, старайтесь заменить его синтаксисом прямого обращения. Тот, кто будет потом сопровождать вашу программу, будет вам только благодарен за это.

Дополнительные переменные экземпляра в подклассах

Одна из замечательных особенностей, присущая хешам, выступающим в качестве хранилища данных, заключается в том, что дочерние классы могут создавать в них дополнительные переменные экземпляра без участия суперкласса. Попробуем от класса `Horse` (лошадь) породить класс `RaceHorse` (скаковая лошадь), который будет обладать теми же свойствами, что и класс `Horse`, но при этом в нем будут иметься переменные для хранения числа первых, вторых и третьих мест, а также количества гонок, в которых лошадь не заняла призового места. Первая часть определения выглядит достаточно тривиально:

```
{ package RaceHorse;
    our @ISA = qw(Horse);
    ...
}
```

Экземпляр класса `RaceHorse` будет инициализироваться значениями, свидетельствующими об отсутствии побед в гонках. Для этого мы перекроем и расширим подпрограмму `named` и добавим четыре дополнительных поля (`первых`, `вторых`, `третьих` и `ни_одного` для числа первых, вторых, третьих и ни одного из вышеперечисленных мест, занятых в гонках):

[1] Неоднозначность проявилась по той причине, что функция `printf()` обычно вызывается для дескриптора файла. Поразмыслив об этой функции, вы наверняка вспомните об отсутствии запятой после дескриптора файла, вследствие чего обращение к этой функции в точности совпадает с синтаксисом косвенного обращения к объекту.

```
{ package RaceHorse;
    our @ISA = qw(Horse);
    ## перекрыть и расширить родительский конструктор:
    sub named {
        my $self = shift->SUPER::named(@_);
        $self->{$_} = 0 for qw(первых вторых третьих ни_одного);
        $self;
    }
}
```

Здесь мы сначала передаем все аргументы конструктору родительского класса, который возвращает полностью сформированный экземпляр класса Horse. Но поскольку при обращении к конструктору передается имя класса RaceHorse, он связывает ссылку именно с классом RaceHorse.[1] Затем добавляются четыре переменные экземпляра, которые отсутствуют в определении суперкласса, и попутно им присваиваются начальные значения. В заключение в вызывающую программу возвращается готовый экземпляр RaceHorse.

Обратите внимание: при создании дочернего класса мы фактически «приоткрыли черный ящик». Нам известно, что в качестве экземпляра суперкласса выступает ссылка на хеш, и что в иерархии суперкласса отсутствуют имена, которые мы использовали для создания переменных в дочернем классе. Мы поступили так потому, что (выражаясь терминологией языка C++ или Java) класс RaceHorse является «дружественным» классом по отношению к классу Horse и имеет прямой доступ к переменным экземпляра. Если программист, сопровождающий классы Animal и Horse, когда-либо внесет в определения классов изменения, ведущие к конфликтам, это может оставаться незамеченным до того момента, как мы откроем свой программный код заказчикам. Еще более интересной может оказаться ситуация, когда ссылка на хеш будет заменена ссылкой на массив.

Один из способов ликвидировать такую зависимость заключается в том, чтобы использовать технику включения родительского класса вместо наследования. В данном примере для этого нужно было бы создать объект Horse как переменную экземпляра класса RaceHorse, а все остальные данные разместить в отдельных переменных экземпляра. Кроме того, потребовалось бы организовать вызов методов класса RaceHorse, унаследованных от класса Horse, через механизм делегирования. Однако, даже несмотря на наличие в языке Perl поддержки всех вышеперечисленных операций, такая реализация оказывается обычно более медлительной и более громоздкой. Но хватит дискуссий.

Далее реализуем несколько методов доступа к переменным:

[1] Аналогичным образом конструктор класса Animal создает экземпляр класса Horse, а не Animal, когда ему в качестве параметра передается имя класса Horse.

```
{ package RaceHorse;
    our @ISA = qw(Horse);
    ## перекрыть и расширить родительский конструктор:
    sub named {
        my $self = shift->SUPER::named(@_);
        $self->{$_} = 0 for qw(первых вторых третьих ни_одного);
        $self;
    }
    sub won { shift->{'первых'}++; }
    sub placed { shift->{'вторых'}++; }
    sub showed { shift->{'третьих'}++; }
    sub lost { shift->{'ни_одного'}++; }
    sub standings {
        my $self = shift;
        join ', ', map "$_: $self->{$_}", qw(первых вторых третьих
                                            ни_одного);
    }
}

my $racer = RaceHorse->named('Билли Бой');
# записать: 3 победы, 1 второе место,
#           1 раз не было занято ни одно призовое место
$racer->won;
$racer->won;
$racer->won;
$racer->showed;
$racer->lost;
print $racer->name, ' занял мест ', $racer->standings, ".\n";
```

В результате:

```
Билли Бой занял мест первых: 1, вторых: 0, третьих: 1, ни_одного: 1.
[объект Билли Бой уничтожен]
[Билли Бой отправлен на фабрику для переработки]
```

Обратите внимание: здесь по-прежнему происходит обращение к деструкторам классов Horse и Animal. Суперклассы понятия не имеют, что мы добавили четыре дополнительных элемента в хеш, тем не менее они вполне работоспособны и исполняют возложенные на них функции.

Переменные класса

Как поступить, если возникнет необходимость обойти всех животных, которые были нами созданы? Животные могут существовать в пространстве имен программы, а могут быть потеряны сразу же после выхода из конструктора named.[1]

Но у нас есть возможность записать созданное животное в хеш, а затем обойти этот хеш в поисках всех животных. Ключами такого хеша мо-

[1] Ну, не совсем потеряны. Это мы можем потерять их, а Perl всегда знает, где они находятся.

гут быть строковые представления ссылок на экземпляры,[1] а значениями – сами ссылки, что даст нам возможность обращаться к ним.

Например, расширим конструктор `named` в классе `Animal`:

```
## в классе Animal
our %REGISTRY;
sub named {
    my $class = shift;
    my $name = shift;
    my $self = { Name => $name, Color => $class->default_color };
    bless $self, $class;
    $REGISTRY{$self} = $self; # возвращает $self
}
```

Имя переменной `%REGISTRY` состоит только из символов верхнего регистра, и это лишний раз напоминает, что она имеет более глобальную область видимости, чем остальные переменные. В данном случае она представляет собой метапеременную, содержащую информацию о множестве экземпляров.

Когда переменная `$self` выступает в качестве ключа, Perl переводит ее содержимое в строковое представление, то есть превращает в строку, уникальную для объекта.

Кроме того, нам необходимо добавить один новый метод:

```
sub registered {
    return map { 'экземпляр '.ref($_)." с именем ".$_->name } values
%REGISTRY;
}
```

Теперь у нас есть возможность получить список всех животных, которые были созданы:

```
my @cows = map Cow->named($_), qw(Бесси Гвен);
my @horses = map Horse->named($_), ('Триггер', 'мистер Эд');
my @racehorses = RaceHorse->named('Билли Бой');
print "Список животных:\n", map(" $_\n", Animal->registered);
print "Конец программы.\n";
```

В результате получим:

```
Список животных:
  экземпляр RaceHorse с именем Билли Бой
  экземпляр Horse с именем мистер Эд
  экземпляр Horse с именем Триггер
  экземпляр Cow с именем Гвен
  экземпляр Cow с именем Бесси
Конец программы
[объект Билли Бой уничтожен]
[Билли Бой отправлен на фабрику для переработки]
```

---

[1] Или любые другие строки, отвечающие требованиям уникальности.

```
[объект мистер Эд уничтожен]
[мистер Эд отправлен на фабрику для переработки]
[объект Триггер уничтожен]
[Триггер отправлен на фабрику для переработки]
[объект Бесси уничтожен]
[объект Гвен уничтожен]
```

Обратите внимание: все животные уничтожаются в порядке уничтожения переменных, которые хранят ссылки на них. Или нет?

Слабые ссылки

Переменная `%REGISTRY` также хранит ссылки на экземпляры животных. Таким образом, даже если программа выйдет из области видимости переменных, содержащих ссылки на этих животных:

```
{
    my @cows = map Cow->named($_), qw(Бесси Гвен);
    my @horses = map Horse->named($_), ('Триггер', 'мистер Эд');
    my @racehorses = RaceHorse->named('Билли Бой');
}
print "Список животных:\n", map(" $_\n", Animal->registered);
print "Конец программы.\n";
```

мы все равно получим тот же результат. Животные не уничтожаются, даже несмотря на то, что переменные, хранившие ссылки на них, покинули область видимости. На первый взгляд этот недостаток можно исправить, изменив деструктор:

```
## в классе Animal
sub DESTROY {
    my $self = shift;
    print '[объект ', $self->name, " уничтожен.]\n";
    delete $REGISTRY{$self};
}
## такое дополнение не решает проблему (читайте дальше)
```

Но это никак не повлияло на результат. Почему? Потому что деструктор не вызывается до тех пор, пока не исчезнет последняя ссылка, но последняя ссылка не может быть уничтожена, пока не будет вызван деструктор.[1]

Одно из решений для достаточно свежих версий Perl[2] состоит в том, чтобы использовать слабые ссылки. *Слабые* ссылки не учитываются механизмом подсчета ссылок. Однако это лучше проиллюстрировать на примере.

---

[1] Мы бы создали тут ссылку на курицу и яйцо, но тогда в классе `Animal` появился бы еще один производный класс.

[2] 5.6 и выше.

Механизм слабых ссылок уже встроен в ядро Perl 5.8. Тем не менее нам нужен внешний интерфейс для доступа к подпрограмме `weaken`, который может быть импортирован из модуля `Scalar::Util`. В Perl 5.6 ту же функциональность можно эмулировать с помощью модуля CPAN – `WeakRef`. После установки модуля (если в этом возникла необходимость)[1] следует обновить конструктор следующим образом:

```
## в классе Animal
use Scalar::Util qw(weaken); # в версии 5.8 и выше
use WeakRef qw(weaken);      # в версии 5.6 после установки модуля из CPAN
sub named {
    ref(my $class = shift) and croak 'допустимо только для имени класса';
    my $name = shift;
    my $self = { Name => $name, Color => $class->default_color };
    bless $self, $class;
    $REGISTRY{$self} = $self;
    weaken($REGISTRY{$self});
    $self;
}
```

Когда Perl подсчитывает количество активных ссылок на «нечто»[2], он не будет учитывать слабые ссылки, преобразованные функцией `weaken`. Если все обычные ссылки были удалены, Perl уничтожает «нечто», а в слабые ссылки записывает значение `undef`.

Теперь все будет работать так, как мы того ожидаем:

```
my @horses = map Horse->named($_), ('Триггер', 'мистер Эд');
print "Список животных перед входом в блок:\n",
      map(" $_\n", Animal->registered);
{
    my @cows = map Cow->named($_), qw(Бесси Гвен);
    my @racehorses = RaceHorse->named('Билли Бой');
    print "Список животных внутри блока:\n", map(" $_\n", Animal->registered);
}
print "Список животных, оставшихся в живых по выходе из блока:\n",
      map(" $_\n", Animal->registered);
print "Конец программы.\n";
```

В результате работы этого примера мы получим:

```
Список животных перед входом в блок:
  экземпляр Horse с именем Триггер
  экземпляр Horse с именем мистер Эд
Список животных внутри блока:
  экземпляр Horse с именем Триггер
  экземпляр Horse с именем мистер Эд
```

1 Порядок установки модулей описывается в главе 3.

2 В документации Perl под «нечто» (thingy) подразумеваются опорные точки ссылок, например объекты. Если вы особенно щепетильны в подборе терминологии, можете называть их объектами ссылок.

```
  экземпляр Cow с именем Бесси
  экземпляр RaceHorse с именем Билли Бой
  экземпляр Cow с именем Гвен
[объект Билли Бой уничтожен.]
[Билли Бой отправлен на фабрику для переработки.]
[объект Гвен уничтожен.]
[объект Бесси уничтожен.]
Список животных, оставшихся в живых по выходе из блока:
  экземпляр Horse с именем Триггер
  экземпляр Horse с именем мистер Эд
Конец программы.
[объект мистер Эд уничтожен.]
[мистер Эд отправлен на фабрику для переработки.]
[объект Триггер уничтожен.]
[Триггер отправлен на фабрику для переработки.]
```

Обратите внимание: скаковая лошадь и коровы прекратили свое существование в конце блока, а обычные лошади в конце программы. Мы добились, чего хотели!

Кроме всего прочего, слабые ссылки могут оказать помощь в ликвидации утечек памяти. Предположим, что нам необходимо вести родословные животных. В объектах животных-родителей можно было бы хранить ссылки на потомков, а в объектах животных-потомков ссылки на родителей.[1]

Мы можем создать слабые ссылки в одном из направлений (или даже в обоих). Если слабые ссылки используются в объектах родителей после того, как все ссылки на объект потомка будут уничтожены, Perl может уничтожить сам объект потомка, а в ссылку объекта-родителя записать значение `undef` (либо в деструкторе можно предусмотреть полное удаление ссылки). При этом объекты животных-родителей не смогут исчезнуть, пока у них есть хотя бы один потомок. Аналогично, если для ссылки на родителей в объектах-потомках будут использоваться слабые ссылки, они просто будут получать значение `undef` после исчезновения объектов-родителей. Это действительно очень гибкое решение.[2]

Без слабых ссылок, как только мы создадим связь родитель-потомок, оба объекта будут оставаться в памяти до окончания работы программы независимо от уничтожения других структур данных, ссылающихся на объекты родителя и потомка.

Будьте внимательны: слабые ссылки следует применять с большой осторожностью, особенно при создании циклических зависимостей. Если данные, на которые имеются слабые ссылки, будут уничтожены

---

[1] Это решает ту проблему циклических ссылок, которую авторы обсуждали несколькими главами ранее. – *Примеч. науч. ред.*

[2] При использовании слабых ссылок необходимо в обязательном порядке проверять их значение, чтобы не разыменовать ссылку со значением `undef`.

раньше времени, это может привести к появлению ошибок, которые очень трудно отыскать и исправить.

Упражнения

Ответы на эти вопросы вы найдете в приложении A в разделе «Ответы к главе 13».

Упражнение [45 мин]

Измените определение класса RaceHorse таким образом, чтобы в момент создания экземпляра информацию об участии в скачках можно было бы извлекать из хеша DBM (где в качестве ключа выступает имя лошади), а при уничтожении экземпляра – сохранять в этом хеше. Например, после четырехкратного запуска следующей программы:

```
my $runner = RaceHorse->named('Билли Бой');
$runner->won;
print $runner->name, ' занял мест ', $runner->standings, ".\n";
```

количество первых мест должно стать равно четырем. При этом экземпляры RaceHorse должны сохранить функциональность, присущую классу Horse.

Для упрощения решения используйте в хеше DBM в качестве значения четыре целых числа, разделенных пробелами.

14

Дополнительные сведения об объектах

У вас, наверное, есть некоторые вопросы, например: «Существует ли некий класс, который является общим для всех объектов?», «Как работает механизм множественного наследования?» или «Как узнать, объект какого класса я использую?». В данной главе мы ответим на эти и некоторые другие вопросы.

Методы класса UNIVERSAL

В процессе создания новых классов мы расширяем их иерархию наследования посредством глобальной переменной `@ISA` в каждом пакете. Чтобы отыскать нужный метод, Perl обходит дерево `@ISA`, пока не обнаружит его или пока не дойдет до конца дерева.

Однако после того как поиск завершится неудачей, Perl всегда просматривает определение класса с именем `UNIVERSAL` и вызывает методы этого класса, если таковые будут обнаружены, как если бы и они находились в определении самого класса или суперкласса.

Класс `UNIVERSAL` можно рассматривать как базовый, от которого порождаются все остальные классы.[1] Если в определение класса поместить некоторый метод, например:

```
sub UNIVERSAL::fandango {
    warn 'объект ', shift, " может станцевать фанданго!\n";
}
```

то все объекты программы смогут вызывать этот метод как `$some_object->fandango`.

---

[1] Точно так же, как, например, в ОО языке Java все объекты являются производными от единого родительского класса `Object`. – *Примеч. науч. ред.*

Вообще мы должны были бы определить метод fandango в требуемых классах, а затем создать определение метода UNIVERSAL::fandango на тот случай, если Perl не сможет отыскать подходящий метод. В качестве практического примера применения класса UNIVERSAL можно было бы привести подпрограммы вывода отладочной информации или сохранения всех объектов программы в файле.[1] Мы определяем некоторый общий метод в классе UNIVERSAL и затем перекрываем его в тех классах, где это необходимо.

Вполне очевидно, что класс UNIVERSAL должен использоваться очень экономно, потому что существует всего один универсальный класс, и наше имя метода fandango может вступить в конфликт с именем fandango из другого модуля, подключенного к программе. По этим причинам класс UNIVERSAL предназначен лишь для определения действительно универсальных методов, вызываемых в процессе отладки или для взаимодействия с внутренними механизмами Perl.

Проверка возможностей объектов

Пакет UNIVERSAL не только обеспечивает место для хранения универсальных методов, но и предоставляет два предопределенных метода: isa и can. Эти методы определены в классе UNIVERSAL, поэтому они доступны всем объектам любых классов.

Метод isa позволяет проверить, является ли текущий класс или экземпляр класса наследником некоторого определенного класса. Рассмотрим семейство классов, порожденных от класса Animal в предыдущих главах:

```
if (Horse->isa('Animal')) { # порожден ли класс Horse от класса Animal?
    print "Класс Horse порожден от класса Animal.\n";
}

my $tv_horse = Horse->named("мистер Эд");
if ($tv_horse->isa('Animal')) { # это класс Animal?
    print $tv_horse->name, " принадлежит классу Animal.\n";
    if ($tv_horse->isa('Horse')) { # это класс Horse?
        print 'Фактически ', $tv_horse->name, " - это класс Horse.\n";
    } else {
        print "...но не принадлежит классу Horse.\n";
    }
}
```

Это очень удобно, когда приходится работать с наборами разнородных объектов и структур данных и при этом отличать категории объектов друг от друга:

```
my @horses = grep $_->isa('Horse'), @all_animals;
```

[1] То, что называется сериализацией. – *Примеч. науч. ред.*

В результате из массива @all_animals в список попадут только объекты, которые принадлежат классу Horse (или RaceHorse). Сравните это со следующей строкой:

```
my @horses_only = grep ref $_ eq 'Horse', @all_animals;
```

которая отберет только экземпляры класса Horse, т. к. оператор ref для экземпляров класса RaceHorse вернет строку, отличную от «Horse».

Вообще мы должны избегать конструкции:

```
ref($some_object) eq 'SomeClass'
```

в программах, поскольку она препятствует будущим пользователям порождать дочерние классы от этого класса. Старайтесь применять метод isa, как это было показано ранее.

Один из недостатков метода isa в данной ситуации заключается в том, что он работает только со ссылками на экземпляры или со скалярами, которые выглядят как имена классов. Если методу передать ссылку, не связанную с экземпляром класса, это вызовет появление фатальной ошибки:

```
Can't call method "isa" on unblessed reference at ...
```

Чтобы избежать этого, метод isa следует вызывать как подпрограмму:

```
if (UNIVERSAL::isa($unknown_thing, 'Animal')) {
    ... это класс Animal! ...
}
```

Ошибки не будет независимо от содержимого переменной $unknown_thing. Но это уход в сторону от объектно-ориентированного стиля, что ведет к появлению других трудностей.[1] Поэтому в таком анализе лучше задействовать механизм исключений, например с помощью функции eval. Если $unknown_thing не является ссылкой, это значит, что мы не должны вызывать метод. Функция eval перехватит ошибку и вернет значение undef, что в данном случае будет означать «ложь»:

```
if (eval { $unknown_thing->isa('Animal') }) {
    ... это класс Animal! ...
}
```

Аналогичным образом предоставляется способ проверить наличие той или иной функциональной возможности с помощью метода can. Например:

```
if ($tv_horse->can('eat')) {
    $tv_horse->eat('сено');
}
```

[1] Если класс Animal определяет собственный метод isa (который, возможно, отбраковывает мутировавших животных), то обращение к подпрограмме UNIVERSAL::isa произойдет в обход Animal::isa и может дать неверный результат.

Если метод can возвращает «истину», значит где-то в иерархии наследования имеется метод eat. Для метода can справедливы те же ограничения, что и для метода isa: $tv_horse должна быть ссылкой на экземпляр класса или скалярной переменной с именем класса. Аналогично может применяться алгоритм обхода ошибочной ситуации:

```
if (eval { $tv_horse->can('eat') } ) { ... }
```

Обратите внимание: если учесть наличие объявленного ранее метода UNIVERSAL::fandango, то

```
$object->can('fandango')
```

всегда будет возвращать значение «истина», поскольку все объекты наследуют функциональность, определенную классом UNIVERSAL.

Метод AUTOLOAD как последняя инстанция

После того как Perl не найдет требуемый метод в иерархии наследования или в классе UNIVERSAL, он не останавливается на достигнутом, а повторяет попытку поиска по той же иерархии (включая и класс UNIVERSAL), пытаясь отыскать метод с именем AUTOLOAD.

Если метод AUTOLOAD существует, он вызывается вместо затребованного метода, и ему передается обычный в таких случаях список входных аргументов: имя класса или ссылка на экземпляр и список аргументов, передававшихся ненайденному методу. Полное имя вызывавшегося, но не найденного метода передается в переменной пакета $AUTOLOAD (в пакете, где данная подпрограмма была скомпилирована), поэтому при необходимости получить простое имя метода придется удалить из имени все символы вплоть до последнего двойного двоеточия.

Подпрограмма AUTOLOAD может сама выполнить все необходимые действия, определить нужную подпрограмму и вызвать ее или аварийно завершить программу, если было запрошено неизвестное действие.

Одно из применений метода AUTOLOAD заключается в том, чтобы отложить компиляцию длинной подпрограммы до момента, когда она действительно станет необходима. Допустим, что метод eat представляет собой очень сложную подпрограмму, которая вызывается далеко не при каждом запуске программы. Тогда, чтобы сэкономить какое-то время, можно было бы отложить ее компиляцию:

```
## в классе Animal
sub AUTOLOAD {
    our $AUTOLOAD;
    (my $method = $AUTOLOAD) =~ s/.*:://s; # удалить имя пакета
    if ($method eq "eat") {
        ## определить метод eat:
        eval q{
            sub eat {
                ...
```

```
                длинное
                тело
                метода
                ...
            }
        }; # конец строки q{ }
        die $@ if $@;      # если вкралась опечатка
        goto &eat;         # вызвать метод
    } else {               # неизвестный метод
        croak "$_[0]: метод $method не найден\n";
    }
}
```

Если зафиксировано обращение к методу `eat` (который оформлен в виде строки, но еще не был скомпилирован), выполняется его компиляция, и затем производится переход к этому методу с помощью специальной конструкции, которая подменяет вызов текущей подпрограммы (`AUTOLOAD`) вызовом метода `eat`, как если бы был вызван метод `&eat`, а не `AUTOLOAD`.[1] После первого обращения к `AUTOLOAD` в определении класса появится новый метод `eat`. Таким образом, в следующий раз этот метод будет вызываться уже напрямую. Такой прием очень хорош для больших программ, т. к. ускоряет их запуск за счет сокращения времени компиляции программного кода.

Автоматизированный способ создания программного кода, который облегчает отключение автозагрузки в процессе разработки и отладки, описан в документации к базовым модулям `AutoLoader` и `SelfLoader`.

Применение AUTOLOAD для реализации методов доступа

В главе 12 было показано, как создать методы доступа к атрибуту цвета животного: `color` и `set_color`. Но если вместо 1–2 атрибутов класс будет иметь 20, тогда программный код будет иметь большое число повторяющихся участков. Однако метод `AUTOLOAD` позволяет сконструировать практически идентичные методы доступа к разным атрибутам, что позволит сэкономить на времени компиляции и уменьшить износ клавиатуры разработчика.

Для реализации такого поведения мы обратимся к ссылкам на подпрограммы. Для начала определим метод `AUTOLOAD` в классе и создадим хеш со списком атрибутов, к которым требуется организовать доступ:

---

[1] Строго говоря, применение оператора `goto` в программах не рекомендуется (и вполне оправданно), но в данном случае это совсем иная форма `goto` – «правильный `goto`», которая как бы меняет имя текущей подпрограммы на заданное. Этот трюк позволяет сделать подпрограмму `AUTOLOAD` вообще невидимой для программы.

```
sub AUTOLOAD {
    my @elements = qw(color age weight height);
```

Затем надо определить тип доступа к заданному атрибуту (чтение или запись), выбрать соответствующий метод и выполнить переход. Вот пример реализации метода чтения :

```
our $AUTOLOAD;
if ($AUTOLOAD =~ /::(\w+)$/ and grep $1 eq $_, @elements) {
    my $field = ucfirst $1;
    {
        no strict 'refs';
        *{$AUTOLOAD} = sub { $_[0]->{$field} };
    }
    goto &{$AUTOLOAD};
}
```

Функция ucfirst задействуется здесь по той простой причине, что метод color извлекает значение атрибута Color. Операция подстановки (символ звездочки) в нашем случае создает подпрограмму, которая возвращает значение требуемого ключа из хеша объекта. Мы не будем углубляться в описание этой части метода. Будем считать, что это магия, и мы просто скопировали и вставили ее в свою программу. В заключение конструкция goto выполняет переход к вновь созданной подпрограмме.

А вот реализация метода записи:

```
if ($AUTOLOAD =~ /::set_(\w+)$/ and grep $1 eq $_, @elements) {
    my $field = ucfirst $1;
    {
        no strict 'refs';
        *{$AUTOLOAD} = sub { $_[0]->{$field} = $_[1] };
    }
    goto &{$AUTOLOAD};
}
```

Если запрошена операция, не соответствующая указанным критериям, просто аварийно завершаем программу:

```
    croak "$_[0] вызван ошибочный метод $method\n";
}
```

Расплачиваться увеличением времени исполнения метода доступа придется только при первом обращении к нему. После этого он уже будет считаться определенной подпрограммой и будет вызываться напрямую.

Более простой способ создания методов доступа

Если способ создания методов доступа на базе метода AUTOLOAD кажется вам непривлекательным, знайте, что на самом деле его лучше не при-

менять, так как в CPAN существует модуль Class::MethodMaker, который позволяет выполнить эти же действия проще.[1]

Упрощенная версия класса Animal может быть определена так:

```
package Animal;
use Class::MethodMaker
    new_with_init => 'new',
    get_set => [-eiffel => [qw(color height name age)]],
    abstract => [qw(sound)],
;
sub init {
    my $self = shift;
    $self->set_color($self->default_color);
}
sub named {
    my $self = shift->new;
    $self->set_name(shift);
    $self;
}
sub speak {
    my $self = shift;
    print $self->name, ': ', $self->sound, "\n";
}
sub eat {
    my $self = shift;
    my $food = shift;
    print $self->name, " ест $food\n";
}
sub default_color {
'коричневый';
}
```

Методы доступа к четырем атрибутам (name, height, color и age) создаются автоматически. Для получения информации о цвете служит метод color, для изменения цвета – set_color. (Флаг eiffel говорит, что «определения методов следует создавать в соответствии с правилами, принятыми в языке программирования Eiffel»). Теперь непривлекательная операция связывания скрыта за ширмой простого метода new. Здесь цвет животного по умолчанию назначается внутри метода init, который вызывается из метода new.

При такой реализации мы по-прежнему можем пользоваться конструктором named, например: Horse->named('мистер Эд'), потому что он сам обращается к подпрограмме new.

Модуль Class::MethodMaker сгенерирует метод sound как абстрактный. Абстрактный метод – это пустой метод, реализация которого должна

[1] В некоторых случаях Class::MethodMaker может оказаться слишком тяжеловесным, тогда можно воспользоваться более легким Class::Accessor.

присутствовать в дочерних классах. Если в дочернем классе метод sound определен не будет, тогда при попытке обращения к нему будет вызван метод, сгенерированный Class::MethodMaker, что приведет к аварийному завершению программы.

Здесь мы утратили возможность обращаться к методам чтения (таким как name) как к методам класса. В свою очередь это не дает возможность обращаться к методам speak и eat класса Animal, как мы делали это ранее. Один из способов решения проблемы заключается в том, чтобы определить более универсальный метод name, который может вызываться и как метод класса, и как метод экземпляра, после чего изменить остальные подпрограммы, обращающиеся к нему:

```
sub generic_name {
    my $either = shift;
    ref $either ? $either->name : "$either без имени ";
}
sub speak {
    my $either = shift;
    print $either->generic_name, ': ', $either->sound, "\n";
}
sub eat {
    my $either = shift;
    my $food = shift;
    print $either->generic_name, " ест $food\n";
}
```

Теперь мы получили определение, которое практически совместимо с предыдущим, за исключением случаев, когда дружественные классы обращались к значениям атрибутов (таким как Color) прямо в хеше, минуя методы доступа ($self->color).

Это снова подводит нас к проблеме сопровождения программного кода. Чем меньше внутренняя реализация класса (тип структуры данных – хеш или массив, имена ключей в хеше или типы элементов) зависит от его интерфейса (имена методов, списки входных аргументов или типы возвращаемых значений), тем более гибкой и более простой в обслуживании будет наша система.

Однако гибкость вовсе не означает высокую производительность. Стоимость вызова метода всегда будет выше стоимости выборки данных из хеша, поэтому в отдельных случаях имеет смысл позволить дружественным классам обращаться к внутренним данным напрямую.

Множественное наследование

Как Perl выполняет обход дерева наследования @ISA? Ответ на этот вопрос и прост, и сложен одновременно. Если механизм множественного наследования не задействуется (то есть массив @ISA содержит всего один элемент), ответ очень прост: Perl переходит от одного @ISA к дру-

гому, пока не найдет базовый класс, массив @ISA которого не содержит элементов.[1]

Алгоритм обхода дерева классов в случае множественного наследования (когда массив @ISA содержит более одного элемента) выглядит значительно сложнее. Предположим, что у нас имеется класс с именем Racer, который реализует базовую функциональность для всего, что может участвовать в гонках, то есть он может служить базовым классом для бегуна, гоночного автомобиля или даже беговой черепахи. При наличии такого класса мы могли бы реализовать класс скаковой лошади так:[2]

```
{
  package RaceHorse;
  our @ISA = qw{ Horse Racer };
}
```

Теперь классу RaceHorse доступно все то, что доступно классу Horse и классу Racer. Когда Perl начинает поиск метода, который отсутствует в реализации самого класса RaceHorse, он сначала обойдет дерево наследования класса Horse (включая все его родительские классы, такие как Animal). Если требуемый метод не будет найден, Perl переключится на класс Racer (и его родительские классы), чтобы определить, не поставляется ли требуемый метод этим классом. Если необходимо, чтобы поиск сначала выполнялся в дереве наследования класса Racer, его имя должно быть указано первым в списке @ISA (рис. 14.1).

Упражнения

Ответы на эти вопросы вы найдете в приложении A в разделе «Ответы к главе 14».

Упражнение 1 [20 мин]

Напишите модуль с именем MyDate, который использовал бы метод AUTOLOAD для обслуживания обращений к методам day, month и year, возвращая соответствующее значение. В случае обращения к неопределенным методам AUTOLOAD должен вызывать подпрограмму carp, чтобы сообщить пользователю о вызове несуществующего метода. Напишите сценарий, который, основываясь на модуле MyDate, выводил бы день, месяц и год.

---

1 Или раньше в этом обходе не встретит класс, содержащий в своем @ISA требуемый метод. – *Примеч. науч. ред.*

2 Если между методами классов Horse и Racer имеется конфликт имен или их реализации не в состоянии обеспечить совместную работу, ситуация может осложниться еще больше. Различные классы в @ISA могут быть плохо совместимы между собой и могут, например, затирать данные друг друга.

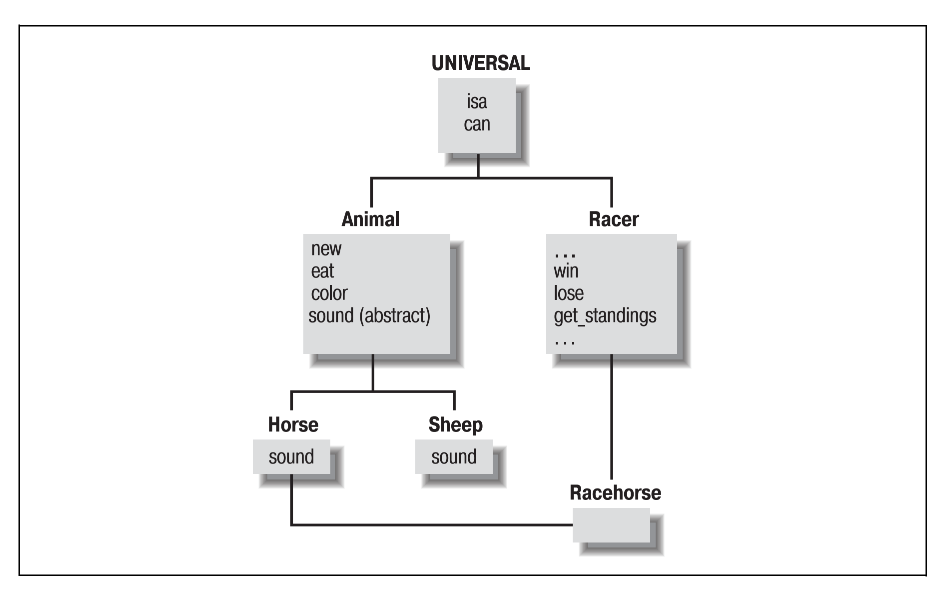


Рис. 14.1. *Класс может не иметь собственные реализации методов, если все, что ему необходимо, он наследует от родительских классов через механизм множественного наследования*

Упражнение 2 [40 мин]

Добавьте в сценарий, который вы написали в первом упражнении, функцию `UNIVERSAL::debug`. Эта функция должна выводить текущее время и текст сообщения, которое должно передаваться ей во входном аргументе. Вызовите метод `debug` объекта `MyDate`. Что произошло? Как это согласуется с механизмом работы `AUTOLOAD`?

15

Экспортирование

В главе 3 было показано, как работать с модулями, которые помещают свои функции в текущее пространство имен. Здесь мы поговорим о том, как заставить свои модули делать то же самое.

Что делает директива use

Так что же в действительности делает директива `use`? Какую роль играет список импортируемых подпрограмм? Список импорта директивы `use` Perl интерпретирует как особую форму блока `BEGIN`, внутри которого находятся директива `require` и вызов одного метода. Например, следующие две операции эквивалентны:

```
use Island::Plotting::Maps qw( load_map scale_map draw_map );

BEGIN {
    require Island::Plotting::Maps;
    Island::Plotting::Maps->import( qw( load_map scale_map draw_map ) );
}
```

Рассмотрим этот программный код по частям. Сначала директиве `require` передается имя пакета, а не строка, как было показано в главе 10. Затем двоеточия в этом имени будут преобразованы в разделители имен каталогов (в UNIX-подобных операционных системах это символ «/»)[1], а к имени будет добавлено расширение `.pm` (от «Perl module»). В UNIX-подобной операционной системе мы в конечном итоге получим:

```
require "Island/Plotting/Maps.pm";
```

Из описания директивы `require`, которое приводилось ранее, следует, что в данном случае Perl будет просматривать список каталогов из мас-

---

[1] Или «\» для ОС Windows. – *Примеч. науч. ред.*

сива @INC в поисках каталога Island, содержащего подкаталог Plotting, внутри которого находился бы файл Maps.pm.[1] Если соответствующий файл не будет найден, на этом работа программы завершится. В противном случае будет загружен и откомпилирован первый встретившийся файл.[2] Как обычно, последнее выражение, вычисленное в найденном модуле директивой require, должно возвращать значение «истина» (иначе программа завершится аварийно)[3]. Как только Perl прочитает файл модуля, он уже не будет повторно считывать его, если где-нибудь еще встретится директива require, требующая загрузки этого же модуля. Предполагается, что загруженный файл определит в качестве интерфейса ряд подпрограмм в своем пространстве имен, а не в пространстве имен вызывающего пакета. Так, если бы мы удалили из файла File::Basename реализацию подпрограмм, то увидели бы примерно следующее:

```
package File::Basename;
sub dirname { ... }
sub basename { ... }
sub fileparse { ... }
1;
```

Эти три подпрограммы определены в пакете File::Basename, а не в пакете, который загрузил их с помощью директивы use. Загруженный файл должен возвращать значение «истина» в последнем выражении; отсюда вполне традиционная последняя строка, содержащая 1;.

Как же имена подпрограмм попадают в пространство имен пакета пользователя? Этот шаг выполняется во второй строке блока BEGIN. Perl автоматически вызывает подпрограмму модуля с именем import, которой передается весь список импорта. Как правило, некоторые из имен в этом списке представляют собой псевдонимы подпрограмм, под которыми их имена импортируются в текущее пространство имен. Ответственность за создание подпрограммы import несет разработчик модуля. Эта подпрограмма выглядит намного проще, чем пояснения, приведенные выше, в чем вы вскоре убедитесь.

И наконец, эта последовательность действий обернута в блок BEGIN. То есть директива use исполняется на этапе компиляции. В результате подпрограммы связаны с соответствующими подпрограммами в модуле, прототипы определены должным образом и так далее.

---

[1] Расширение .pm определяется интерфейсом Perl и не может быть изменено. Таким образом, все имена файлов с модулями должны оканчиваться расширением .pm.

[2] Удовлетворивший всем этим условиям: имеется в виду, что массив массива @INC может содержать (далее) другие каталоги, в которых может присутствовать Island/Plotting/Maps.pm. – *Примеч. науч. ред.*

[3] Эту ошибку можно перехватить с помощью функции eval.

Импорт с помощью модуля Exporter

В главе 3 мы перепрыгнули через последовательность действий, где подпрограмма import (созданная автором модуля) должна взять имя File::Basename::fileparse и каким-то образом определить ее псевдоним в вызывающем пакете, чтобы она стала доступна под именем fileparse.

Perl предоставляет ряд конструкций для проведения интроспекции (анализа внутреннего строения программы). Например, мы можем просмотреть таблицу символов (где сосредоточены все имена подпрограмм и переменных), увидеть, какие из них определены, и изменить эти определения. Кое-что вы уже видели, когда рассматривали механизм работы метода AUTOLOAD в главе 14. Если бы мы были авторами модуля File::Basename и хотели бы просто перенести имена подпрограмм filename, basename и fileparse из текущего пакета в пакет main, то могли бы написать подпрограмму import так:

```
sub import {
    no strict 'refs';
    for (qw(filename basename fileparse)) {
        *{"main::$_"} = \&$_;
    }
}
```

Прямо ребус какой-то! Да еще и с ограничениями. А что если вызывающий пакет не хочет импортировать подпрограмму fileparse? Что если вызывающий пакет называется вовсе не main?

К счастью, в модуле Exporter есть стандартная функция import. Чтобы воспользоваться ею, автору модуля достаточно включить строку

```
use base qw(Exporter);
```

Теперь обращение к функции import будет передано классу Exporter, который знает, как обработать список импортируемых подпрограмм[1] и экспортировать их в вызывающий пакет.

@EXPORT и @EXPORT_OK

Функция import, предоставляемая модулем Exporter, проверяет значение переменной @EXPORT модуля, чтобы определить перечень имен, экспортируемых по умолчанию. Так, модуль File::Basename мог бы сделать примерно следующее:

```
package File::Basename;
our @EXPORT = qw( basename dirname fileparse );
use base qw(Exporter);
```

Список @EXPORT определяет как перечень подпрограмм, доступных для экспорта (общедоступный интерфейс), так и список подпрограмм, экс-

[1] И переменных, хотя мало кто так делает, и вряд ли надо подражать им.

портируемых по умолчанию, если не определен список импорта. Например, следующие две строки можно считать эквивалентными:

```
use File::Basename;

BEGIN { require File::Basename; File::Basename->import }
```

Функции import список не передается. В этом случае подпрограмма Exporter->import прочитает содержимое массива @EXPORT и импортирует все подпрограммы из этого списка.[1]

Как быть, если мы не хотим включать некоторые подпрограммы в список по умолчанию, но при этом хотим, чтобы они были доступны для импорта? Имена этих подпрограмм можно добавить в список @EXPORT_OK пакета модуля. Предположим, что модуль Джиллигана по умолчанию экспортирует подпрограмму guess_direction_toward, но при этом он может экспортировать подпрограммы ask_the_skipper_about и get_north_from_professor, если это будет явно запрошено. Тогда модуль может начинаться со строк:

```
package Navigate::SeatOfPants;
our @EXPORT = qw(guess_direction_toward);
our @EXPORT_OK = qw(ask_the_skipper_about get_north_from_professor);
use base qw(Exporter);
```

В этом случае допустимыми были бы следующие обращения к директиве:

```
use Navigate::SeatOfPants; # импортирует guess_direction_toward

use Navigate::SeatOfPants qw(guess_direction_toward); # то же самое

use Navigate::SeatOfPants
    qw(guess_direction_toward ask_the_skipper_about);

use Navigate::SeatOfPants
    qw(ask_the_skipper_about get_north_from_professor);
    ## НЕ импортирует подпрограмму guess_direction_toward!
```

Если мы указываем какое-либо имя, оно должно присутствовать либо в списке @EXPORT, либо в списке @EXPORT_OK. Таким образом, следующий запрос будет отклонен функцией Exporter->import:

```
use Navigate::SeatOfPants qw(according_to_GPS);
```

поскольку подпрограмма according_to_GPS не определена ни в одном из списков.[2] Благодаря этим двум спискам мы можем управлять общедоступным интерфейсом модуля. Но это не помешает любому желающему обратиться к подпрограмме Navigate::SeatOfPants:according_to_GPS (если

---

[1] Не забывайте, пустой список импорта и его отсутствие – это не одно и то же. Если список импорта пуст, функция import просто не вызывается.

[2] Под эту проверку попадают также опечатки и ошибочные имена подпрограмм, и поэтому очень просто понять, почему, например, не работает подпрограмма get_direction_from_professor.

таковая существует), но, по крайней мере, они будут знать, что мы не собирались предложить эту подпрограмму для общего пользования.

%EXPORT_TAGS

Не обязательно включать в список все функции и переменные, которые будут импортироваться модулем. Достаточно создать ярлыки, или теги, и затем сгруппировать их под одним именем. В списке импорта перед тегом необходимо добавлять символ двоеточия. Например, в базовом модуле `Fcntl` определены несколько констант `flock`, доступные как группа с тегом `:flock`:

```
use Fcntl qw( :flock ); # импорт всех констант flock
```

В документации к модулю `Exporter` оговаривается, что некоторые теги уже определены по умолчанию. Тег `DEFAULT` представляет группу имен подпрограмм и переменных, импортируемых из модуля по умолчанию, как если бы список импорта вообще не был указан:

```
use Navigate::SeatOfPants qw(:DEFAULT);
```

В таком использовании тега `DEFAULT` нет большого смысла, но если необходимо в дополнение к именам по умолчанию импортировать еще что-нибудь, можно не вводить весь список, а просто указать:

```
use Navigate::SeatOfPants qw(:DEFAULT get_north_from_professor);
```

Однако такой прием достаточно редко встречается на практике. Почему? Дело в том, что прямое объявление списка импорта обычно служит для управления именами подпрограмм, вызываемых из программы. Последние примеры не гарантируют неизменность списка имен подпрограмм при переходе на более новую версию модуля, где в список по умолчанию могут быть добавлены дополнительные имена, которые в свою очередь могут вступать в конфликт с именами в самой программе.[1] В некоторых случаях модуль может поставлять десятки и даже сотни имен. В таких модулях с целью облегчения импорта совокупностей имен применяются более совершенные приемы (описанные в документации к модулю `Exporter`).

Для определения имен тегов в модулях можно использовать хеш `%EXPORT_TAGS`. В качестве ключей хеша выступают имена тегов (без двоеточия), а в качестве значений – анонимные массивы имен.

```
package Navigate::SeatOfPants;
use base qw(Exporter);
```

[1] По этой же причине считается дурным тоном в новых версиях модуля вводить дополнительные имена в список импорта по умолчанию. Тем не менее, если известно, что с момента выпуска первой версии некоторая функция до сих пор не реализована, нет причин, по которым нельзя было бы вставить функцию-заглушку, например `sub according_to_GPS {die "еще не реализована"}`.

```
our @EXPORT = qw(guess_direction_toward);
our @EXPORT_OK = qw(
                get_north_from_professor
                according_to_GPS
                ask_the_skipper_about
                );

our %EXPORT_TAGS = (
              all => [ @EXPORT, @EXPORT_OK ],
              gps => [ qw( according_to_GPS ) ],
        direction => [ qw(
                get_north_from_professor
                according_to_GPS
                guess_direction_toward
                ask_the_skipper_about
                ) ],
        );
```

Первый тег здесь, `all`, включает все экспортируемые имена (входящие в состав списков `@EXPORT` и `@EXPORT_OK`). Тег `gps` включает в себя только подпрограммы, предназначенные для работы с GPS (Global Positioning System – глобальная система позиционирования). Тег `direction` включает в себя все подпрограммы, связанные с определением направления движения. Теги могут содержать повторяющиеся имена (обратите внимание: подпрограмма `according_to_GPS` включена в состав всех трех тегов). Независимо от того, как будут описаны теги, все имена, которые они включают, должны находиться либо в списке `@EXPORT`, либо в списке `@EXPORT_OK`.

После того как будут определены теги экспорта, пользователь сможет определить по ним список импорта:

```
use Navigate::SeatOfPants qw(:direction);
```

Экспорт имен в объектно-ориентированных модулях

Как уже отмечалось, использование объектно-ориентированных модулей обычно заключается в обращении к методам класса и методам экземпляров класса, созданных с помощью конструкторов. Это подразумевает, что объектно-ориентированные модули, как правило, ничего непосредственно не экспортируют. Таким образом:

```
package My::OOModule::Base;
our @EXPORT = ( ); # эта строка может быть опущена
use base qw(Exporter);
```

Как же породить дочерний класс от этого класса? Прежде всего необходимо помнить, что метод `import` определен в классе `Exporter`, поэтому в начало пакета следует вставить примерно следующие строки:

```
package My::OOModule::Derived;
use base qw(Exporter My::OOModule::Base);
```

А разве вызов `My::OOModule::Derived->import` не приведет в конечном счете к вызову метода класса `Exporter` через `My::OOModule::Base`? Разумеется, приведет. Поэтому упоминание класса `Exporter` можно опустить:

```
package My::OOModule::Derived;
use base qw(My::OOModule::Base);
```

Упоминание класса `Exporter` необходимо только в базовом классе, который служит основанием иерархии при условии, что он не наследует какие-либо другие классы.

Постарайтесь ознакомиться со списком зарезервированных имен методов (на страницах справочного руководства к классу `Exporter`), которые нельзя использовать для именования методов в своих объектно-ориентированных модулях. К моменту написания этих строк список зарезервированных имен включал в себя `export_to_level`, `require_version` и `export_fail`. Кроме того, желательно избегать имя `unimport`, поскольку Perl вызывает эту подпрограмму, когда директива `use` заменяется на `no`. Это не часто встречается в модулях, написанных пользователями, но не редкость в таких директивах, как `strict` и `warnings`.

Даже притом, что объектно-ориентированные модули ничего не экспортируют, иногда бывает желательно экспортировать имена конструкторов или подпрограмм обслуживания модуля. Такие подпрограммы, как правило, работают как методы класса, но желательно, чтобы они были доступны пользователю как обычные подпрограммы.

Рассмотрим в качестве примера библиотеку `LWP`, которая доступна в CPAN как составная часть дистрибутива libwww-perl. Ныне модуль `URL::URL` считается устаревшим и был заменен модулем `URI`, предназначенным для работы с универсальными идентификаторами ресурсов, которые обычно выглядят как обычные URL, например *http://www.gilligan.crew.hut/maps/island.pdf*. Можно создать объект `URI::URL` посредством традиционного конструктора:

```
use URI::URL;
my $u = URI::URL->new('http://www.gilligan.crew.hut/maps/island.pdf');
```

Однако в список импорта модуля `URI::URL` входит подпрограмма `url`, которая также может применяться в качестве конструктора:

```
use URI::URL;
my $u = url('http://www.gilligan.crew.hut/maps/island.pdf');
```

Поскольку данная импортируемая подпрограмма не является методом класса, при обращении к ней можно обойтись без оператора «стрелка». Кроме того, подпрограмма отличается от всего остального, что находится в модуле, тем, что не получает имя класса в качестве первого аргумента. Поэтому, когда в пакете присутствуют как методы, так

и обычные подпрограммы, автор модуля должен оговорить порядок обращения к ним.

Вспомогательная подпрограмма url присутствовала в пакете с самого начала. Однако она вступала в конфликт с одноименной подпрограммой, экспортируемой модулем CGI.pm, что приводило к появлению весьма интересных ошибок (особенно при настройке mod_perl). Модуль, который подгружался последним, выигрывал схватку. (В современном интерфейсе модуля URI эта функция не экспортируется.) Ранее, чтобы избежать аварийного завершения программы, приходилось указывать пустой список импорта:

```
use URI::URL ( ); # не импортировать подпрограмму "url"
my $u = URI::URL->new(...);
```

Собственные подпрограммы импорта

В качестве примера рассмотрим реализацию подпрограммы импорта в модуле CGI.pm, после чего попробуем создать свою собственную подпрограмму. Не удовлетворенный невероятной гибкостью подпрограммы import из модуля Exporter, автор модуля CGI.pm Линкольн Стейн (Lincoln Stein) создал собственную подпрограмму import специально для модуля CGI.pm.[1] Если вам когда-нибудь придется познакомиться с необычайно богатым набором необязательных параметров, которые могут идти вслед за use CGI, вы наверняка зададитесь вопросом – как же все это работает? В этом нет ничего сложного – это лишь вопрос терпения программиста. Кстати, вы можете исследовать исходные тексты модуля самостоятельно.

Данная версия функции import позволяет работать с модулем CGI как с объектно-ориентированным:

```
use CGI;
my $q = CGI->new;          # создать объект запроса
my $f = $q->param('foo'); # получить значение поля foo
```

и как с процедурно-ориентированным модулем:

```
use CGI qw(param);    # импортировать подпрограмму param
my $f = param('foo'); # получить значение поля foo
```

Если не хотите указывать все возможные импортируемые подпрограммы, можете импортировать их так:

```
use CGI qw(:all); # определить подпрограмму 'param' и еще 800 подпрограмм
my $f = param('foo');
```

---

1 Некоторые разработчики использовали эту подпрограмму (известную как «Lincoln Loader» – «загрузчик Линкольна») из глубокого уважения к автору и одновременно из страха перед необходимостью иметь дело с чем-то, что просто работает не так, как то, с чем им приходилось встречаться раньше.

Есть еще целый ряд вариантов применения дополнительных параметров. Например, при желании можно запретить обработку полей с запоминанием их содержимого, для этого достаточно добавить параметр `-nosticky` в список импорта:

```
use CGI qw(-nosticky :all);
```

Можно создать собственные подпрограммы `start_table` и `end_table` в дополнение к другим, записав:

```
use CGI qw(-nosticky :all *table);
```

Список возможных параметров потрясает своим разнообразием. Как же Линкольну это удалось? Вы можете просмотреть исходные тексты модуля `CGI.pm`, чтобы увидеть все своими глазами, а здесь мы покажем лишь самые основные моменты.

Метод `import` – это обычный метод. Таким образом, его можно заставить сделать все, что только душе угодно. Мы уже показывали реализацию простой функции импорта (хотя и гипотетической) на примере модуля `File::Basename`. Теперь мы обойдемся без функции `import` из модуля `Exporter` и напишем собственную, которая будет экспортировать имена в пространство имен `main`.

```
sub import {
    no strict 'refs';
    for (qw(filename basename fileparse)) {
        *{"main::$_"} = \&$_;
    }
}
```

Такой прием эффективен только в том случае, если вызывающий пакет имеет имя по умолчанию `main`, так как это имя жестко определено в подпрограмме. Но у нас есть возможность выяснить имя пакета «на лету» с помощью встроенной подпрограммы `caller`. В скалярном контексте эта подпрограмма возвращает имя вызывающего пакета:

```
sub import {
    no strict 'refs';
    my $package = caller;
    for (qw(filename basename fileparse)) {
        *{$package . "::$_"} = \&$_;
    }
}
```

В списочном контексте подпрограмма `caller` возвращает гораздо больше информации.

```
sub import {
    no strict 'refs';
    my( $package, $file, $line ) = caller;
    warn "вызов из пакета $package, в файле $file\n";
    for (qw(filename basename fileparse)) {
```

```
        *{$package . "::$_"} = \&$_;
    }
}
```

Поскольку подпрограмма `import` – это самый обычный метод, любые аргументы, получаемые им, находятся в массиве `@_` (если вы еще помните, это список импортируемых имен). Можно проанализировать этот список и решить, что делать с каждым отдельным элементом. Давайте включим отладочный режим, если в списке импорта будет обнаружен параметр `debug`. Мы не будем импортировать подпрограмму с таким именем, а просто запишем в переменную `$debug` значение «истина», а затем произведем те же действия, что и раньше. Но на этот раз предупреждение будет выдаваться, только если был включен режим отладки.

```
sub import {
    no strict 'refs';
    my $debug = grep { $_ eq 'debug' } @_;
    my( $package, $file, $line ) = caller;
    warn "вызов из пакета $package, в файле $file\n" if $debug;
    for (qw(filename basename fileparse)) {
        *{$package . "::$_"} = \&$_;
    }
}
```

К аналогичным трюкам Линкольн прибегал при разработке модуля `CGI`. Практически то же самое мы увидим в подпрограмме `import` модуля `Test::More`, которая управляет порядком тестирования, когда будем рассматривать этот модуль в главе 17.

Упражнения

Ответы на эти вопросы вы найдете в приложении A в разделе «Ответы к главе 15».

Упражнение 1 [15 мин]

Возьмите библиотеку `Oogaboogoo`, созданную вами в упражнении 1 главы 10, и превратите ее в модуль, который можно будет подключать с помощью директивы `use`. Измените программный код, обращающийся к библиотеке, таким образом, чтобы он мог использовать импортированные подпрограммы (без указания полного имени подпрограмм, включающих имя модуля) и проверьте его.

Упражнение 2 [15 мин]

Измените предыдущий пример таким образом, чтобы имелась возможность указать тег экспорта `all`. В случае, когда указывается тег `all`, ваш модуль должен импортировать все подпрограммы, имеющиеся в библиотеке.

```
use Oogaboogoo::date qw(:all);
```

16

Создание дистрибутива

В главе 15 рассказывалось о вымышленном модуле `Island::Plotting::Maps` и о том, как добавить в него поддержку импорта с помощью модуля `Exporter`, чтобы этот модуль можно было подключить к программе посредством директивы `use`.

Файл `.pm` был нам полезен, но работать с ним неудобно. Прежде чем отдать плоды своего труда другим разработчикам, которые могли бы установить наш модуль на своих машинах, необходимо выполнить еще целый ряд действий.

Каталог установки

Как сделать, чтобы Perl смог найти модуль? Где в системе должен находиться программный код? Смогут ли пользователи, не обладающие привилегиями администратора системы, установить модуль в каталог, где он будет доступен всем пользователям, или они должны будут устанавливать модуль в свой каталог?

Документация

Где должна находиться документация модуля? Как устанавливать документацию, чтобы сделать ее доступной пользователям?

Полнота архива

Следует ли включать в состав архива дополнительное ПО, в котором будут нуждаться пользователи? Что еще надо приложить к программному коду модуля, чтобы сделать его пригодным для использования?

Тестирование

Как узнать, правильно ли работает программное обеспечение? Уверены ли мы, что оно делает все, что нам нужно? Как убедиться, что оно работает одинаково в разных операционных системах и с разными версиями Perl?

Интерфейсы на языке С

Как правильно описать порядок компиляции и сборки программного кода в окружающей среде разработчика или конечного пользователя, если в состав модуля входит программный код на языке С (об этом здесь говориться не будет)?

Собрать дистрибутив можно разными способами

Дистрибутив содержит в себе модуль (или набор взаимосвязанных модулей) и дополнительные файлы с документацией, тестами и инструкциями по установке модуля. Все эти файлы можно создать вручную, но гораздо удобнее воспользоваться программным обеспечением, которое все это сделает за нас.

В самом начале[1] дистрибутивы приходилось создавать вручную. Некоторое время спустя появилась программа *h2xs*, которая автоматизировала процесс сборки дистрибутивов, и ее возможностей вполне хватало до некоторого времени. Затем Энди Лестер (Andy Lester) создал модуль `Module::Starter`, предназначенный для создания дистрибутивов, Джим Кинан (Jim Keenan) разработал модуль `ExtUtils::ModuleMaker`, который улучшил программу *h2xs*, а также появилась масса других модулей, способных упростить процесс создания дистрибутивов.[2]

Неважно, каким способом создается дистрибутив или какие инструментальные средства при этом применяются, сам процесс остается неизменным. Мы запускаем сценарий на языке Perl, который создает файл, куда помещает все необходимое для подготовки и установки программного кода. С его помощью можно протестировать и установить модуль.

В случае использования традиционного *Makefile.PL*, процесс установки начинается с команды:

```
$ perl Makefile.PL
```

После этого можно проверить и установить модуль, дав команду *make* и указав перечень действий, которые необходимо выполнить:[3]

```
$ make all test install
```

Можно отказаться от файла *Makefile.PL*. Он предполагает наличие внешней программы *make*, которая изначально появилась как инструмент операционной системы UNIX и потому может отсутствовать в дру-

---

1 По крайней мере, когда Perl 5 только появился, еще до выхода в свет системных модулей сторонних разработчиков.

2 Брайан просто создает каталог шаблона с помощью `ttree` из модуля `Template::Toolkit`. Об этой методике можно прочитать в декабрьском выпуске «The Perl Journal» за 2004 год: *http://www.tpj.com/documents/s=9622/tpj0412e/0412e.html*.

3 Действия `test` и `install` можно выполнить за одно обращение к утилите *make*. Если в процессе работы будут обнаружены ошибки, она прервет установку.

гих операционных системах.[1] Кеном Вильямсом (Ken Williams) был разработан модуль `Module::Build`, способный заменить *make*. Так как заранее известно, что перед установкой модуля Perl уже установлен, мы вполне можем использовать его для установки других модулей.

Процесс установки с помощью модуля `Module::Build` выглядит практически точно так же, за исключением того, что в этом случае задействуется файл `Build.PL`.

```
$ perl Build.PL
```

После этого выполняются почти те же самые действия, что и прежде.

```
perl Build
perl Build test
perl Build install
```

При выборе того или иного способа обычно исходят из следующих соображений. Метод на основе *Makefile.PL* применяется уже достаточно давно и корректно работает практически всегда за редким исключением. Этот метод основан на использовании модуля `ExtUtils::Makemaker`, который поставляется в составе дистрибутива Perl. К сожалению, `ExtUtils::Makemaker` превратился в неповоротливого монстра, сложного для поддержки из-за необходимости учитывать особенности множества операционных систем, в которых работает Perl. Метод на базе модуля `Module::Build` более современный, но еще не получил широкого распространения.[2] Однако он не требует наличия внешних инструментальных средств, и поэтому многим будет проще работать с ним.

Программа h2xs

Здесь мы опишем метод на базе файла *Makefile.PL* и программы со странным именем *h2xs*.[3] С помощью утилиты *h2xs* мы создадим фай-

---

1 В ОС Windows существует утилита `nmake`, разработанная в корпорации Microsoft.

2 И все же `Module::Build` войдет в состав стандартного дистрибутива Perl начиная с версии 5.10 и, скорее всего, станет стандартным способом установки будущих версий других модулей.

3 Интересна история имени *h2xs*. В самом начале, как только появился Perl 5, Ларри придумал язык XS для описания промежуточного программного кода, с помощью которого из сценариев на языке Perl можно было бы обращаться к функциям из библиотек, написанных на языке C. Изначально этот программный код был написан вручную, а позднее была разработана утилита *h2xs*, которая просматривала заголовочные файлы на языке C (файлы с расширением *.h*) и создавала соответствующий им код на языке XS. Отсюда и название *h 2* (to) *XS*. С течением времени в программу было добавлено множество разнообразных функций, включая создание файлов шаблонов дистрибутивов. В этой главе мы будем рассматривать порядок использования *h2xs* для создания дистрибутивов, которые никакого отношения не имеют ни к заголовочным файлам, ни к языку XS. Поразительно.

лы шаблонов, которые будут служить отправной точкой для создания дистрибутива. Для этого достаточно дать команду `h2xs -XAn` и передать ей имя модуля – в данном случае `Island::Plotting::Maps`.[1] В результате на экране мы получим примерно такой вывод:[2]

```
Defaulting to backwards compatibility with perl 5.8.7
If you intend this module to be compatible with earlier perl versions, please
specify a minimum perl version with the -b option.

Writing Island-Plotting-Maps/lib/Island/Plotting/Maps.pm
Writing Island-Plotting-Maps/Makefile.PL
Writing Island-Plotting-Maps/README
Writing Island-Plotting-Maps/t/Island-Plotting-Maps.t
Writing Island-Plotting-Maps/Changes
Writing Island-Plotting-Maps/MANIFEST
```

Файл MANIFEST

Программа *h2xs* создала каталог и несколько файлов в нем. Это практически те же файлы, которые создаются другими инструментальными средствами, предназначенными для создания дистрибутивов. Даже если вы откажетесь от программы *h2xs*, вам все равно необходимо будет понимать содержимое этих файлов. Перейдем к рассмотрению файла *MANIFEST*, в котором содержится список файлов дистрибутива. Файл *MANIFEST* начинается со строк:[3]

```
Changes
Makefile.PL
MANIFEST
README
t/Island-Plotting-Maps.t
lib/Island/Plotting/Maps.pm
```

Файл манифеста на самом деле представляет собой оглавление дистрибутива. Когда наступит время создания архива, в его состав будут включены все файлы, перечисленные в манифесте. Когда дистрибутив будет устанавливаться конечным пользователем, программа установки проверит наличие в архиве всех файлов, перечисленных в манифесте. Собственно, все уже готово к созданию архива. Для этого сначала следует запустить сценарий *Makefile.PL*. После этого в каталоге появится новый файл *Makefile*.

```
$ perl Makefile.PL
$ make tardist
```

---

[1] Если в дистрибутив включаются несколько модулей, то в команде надо указать имя главного модуля. Остальные можно добавить позднее.

[2] У вас может получиться не совсем то же самое, например из-за различия версий Perl.

[3] Содержимое файла у вас также может отличаться в зависимости от версии Perl.

Если вместо архива `tar` надо создать `zip`, то замените последнюю команду на:

```
$ make zipdist
```

Сопровождение файла манифеста может показаться делом достаточно сложным, но мы уверены, что в дистрибутив не попадут случайные файлы, которые просто оказались «не в то время не в том месте». Позднее будет показано, как обновить файл манифеста.

Файл README

Следующий файл – это файл *README*. Стандартный файл, в котором содержится краткая информация, необходимая пользователям для беглого знакомства с модулем. Общепринято включать в этот файл, по крайней мере, краткое описание модуля, инструкции по установке и сведения о порядке лицензирования.

```
Island-Plotting-Maps version 0.01
=================================

Файл README содержит краткое описание модуля, инструкции по установке,
перечень зависимостей, которые могут быть (например, требования к наличию
компилятора языка C и установленных библиотек), и любые другие сведения,
которые необходимо знать прежде, чем модуль будет установлен.

Файл README обязательно должен входить в состав модулей, распространяемых
через CPAN, поскольку в CPAN файл README будет извлечен из дистрибутива
модуля и выложен для просмотра потенциальными пользователями модуля.
Как правило, этот файл должен содержать информацию о версии модуля, чтобы
пользователи могли принять решение, стоит ли загружать и устанавливать
этот модуль у себя.

УСТАНОВКА

Чтобы установить этот модуль необходимо выполнить
следующую последовательность команд:

  perl Makefile.PL
  make
  make test
  make install

ЗАВИСИМОСТИ

Этот модуль требует наличия следующих модулей и библиотек:

  бла бла бла

АВТОРСКИЕ ПРАВА И ЛИЦЕНЗИЯ

Здесь должна находиться информация об авторских правах.

Copyright (C) 2005 by Ginger Grant
```

```
Данная библиотека является свободно распространяемым программным
обеспечением.
Допускается  ее свободное распространение и/или изменение на тех же условиях,
что и сам Perl, либо Perl версии 5.8.7 или любой более поздней версии Perl 5,
которая может быть установлена у вас.
```

Вполне очевидно, что этот файл придется отредактировать, чтобы включить в него все, что вы захотите сообщить. Фраза «бла-бла-бла» (blah blah blah) очень часто выступает в качестве заполнителя и впоследствии заменяется реальным текстом.[1] Если подобные шаблоны, созданные программой *h2xs*, оставить без изменения, у потенциального пользователя появятся подозрения, что и сам программный код модуля содержит массу ошибок и неточностей. Поэтому подобные шаблоны необходимо отредактировать (как и программный код), прежде чем выпустить дистрибутив в свет.

Особое внимание обращайте на раздел с описанием авторских прав и лицензии. (Здесь должно стоять ваше имя, а не «Джинджер» (Ginger), если, конечно, ваша машина точно знает, кто сидит за ее клавиатурой.) Ваш заказчик может потребовать изменить указание на авторские права, чтобы, например, включить название своей компании, а не ваше имя. Если вы включаете в состав дистрибутива чужой программный код, то, может быть, надо упомянуть и об авторских правах на него.

Кроме того, файл *README* имеет специальное назначение: в CPAN этот файл автоматически извлекается из архива и передается поисковым системам различных всемирных архивов для индексирования. Инструментальные средства для работы с CPAN позволяют загружать файлы *README*, не загружая при этом сами дистрибутивы. Например, в командной оболочке `CPAN.pm` можно дать следующую команду:[2]

```
$ perl -MCPAN -eshell
cpan> readme Island::Plotting::Maps
```

В модуле `CPANPLUS`, преемнике `CPAN.pm`, есть аналогичные команды:

```
$ perl -MCPANPLUS -eshell
cpanp> r Island::Plotting::Maps
```

В зависимости от версии Perl у вас может быть установлена программа *cpan* или *cpanp*:

```
$ cpan
cpan> readme Island::Plotting::Maps
```

---

[1] Ради интереса можете попробовать поискать в CPAN все места, где встречается фраза `blah blah blah`.

[2] Вы могли бы сделать это, если бы модуль `Island::Plotting::Maps` действительно находился в CPAN.

Файл Changes

В файле *Changes* содержится история развития модуля. Обычно пользователи просматривают этот файл, чтобы узнать о появившихся нововведениях и решить, стоит ли обновлять свою версию.[1] В этом файле должна указываться информация об обнаруженных и исправленных ошибках, о дополнительных функциональных возможностях и о степени важности внесенных изменений (то есть обязательно ли всем, кто пользуется модулем, обновлять версию).

```
$ cat Changes
Revision history for Perl extension Island::Plotting::Maps.

0.01 Wed Oct 16 15:53:23 2002
        - original version; created by h2xs 1.22 with options
                -XAn Island::Plotting::Maps
```

Этот файл должен поддерживаться и изменяться вручную, если среда разработки не предоставляет инструментальных средств, способных делать это автоматически. Он может использоваться для отслеживания изменений и их влияния на появление ошибок: если известно, что некоторая ошибка проявилась тремя выпусками раньше, можно будет попробовать проанализировать ее взаимосвязь с теми новшествами, что появились в то время.

Файл META.yml

Самые свежие версии инструментальных модулей создают в дистрибутиве файл *META.yml*, где в удобочитаемом виде содержится информация о модуле. Сведения в этом файле представлены в формате YAML, о котором уже говорилось в главе 6.

```
# http://module-build.sourceforge.net/META-spec.html
#XXXXXXX This is a prototype!!! It will change in the future!!! XXXXX#
name:         HTTP-Size
version:      0.91
version_from: lib/Size.pm
installdirs:  site
requires:
        HTML::SimpleLinkExtor:        0
        HTTP::Request:                0
        URI:                          0

distribution_type: module
generated_by: ExtUtils::MakeMaker version 6.17
```

---

[1] Почему может возникнуть желание отказаться от обновления версии? Многих вполне может удовлетворять (и вполне оправданно) то, чем они уже обладают. Зачем тогда рисковать и загружать новую версию, в которой могут содержаться новые ошибки?

Файл с таким содержимым был у меня создан модулем `ExtUtils::MakeMaker`. В процессе создания архива дистрибутива этот файл будет автоматически создан сценарием *Makefile*, добавлен в список в файле *MANIFEST* и включен в состав дистрибутива. Другие инструментальные средства, о которых будет рассказано в главе 19, смогут с помощью этого файла извлечь необходимые сведения о модуле, избежав необходимости запускать какой-либо программный код из дистрибутива.

Прототип модуля

Наконец мы подошли к самой важной части дистрибутива – к программному коду.

```
package Island::Plotting::Maps;
```

Пока все выглядит неплохо. Модуль начинается с необходимой директивы `package`. За ней следуют стандартные директивы.

```
use 5.008007;
use strict;
use warnings;
```

Здесь объявлено, что модуль совместим с Perl 5.8.7 или более поздней версией и что ограничения компилятора и вывод предупреждений разрешены по умолчанию. Мы приветствуем подобную практику. Разумеется, эти строки могут быть удалены или изменены. Если известно, что модуль сможет работать под управлением Perl более ранней версии, можно соответствующим образом изменить строку `use 5.008007;` (или вообще удалить ее).

Далее следуют строки, которые выполняют экспорт функций и переменных модуля в пакет, подключающий данный модуль. Все необходимые действия будут выполнены модулем `Exporter`, предоставляющим метод `import`, о чем рассказывалось в главе 15.

```
require Exporter;

our @ISA = qw(Exporter);
```

Эти строки типичны для необъектно-ориентированных модулей, поскольку объектно-ориентированные модули ничего не экспортируют – принцип их действия основан на обращении к методам класса. Таким образом, мы заменим строки, связанные с подключением модуля `Exporter`, на строки объявления базового класса, который наследуется нашим классом, если таковой существует.

```
use base qw(Geo::Maps);
```

Затем мы должны добавить некоторые сведения, необходимые для класса `Exporter`, а все остальное сделает программа *h2xs*. Нам надо определить (можно и не определять) три переменные:

```
our %EXPORT_TAGS = ( 'all' => [ qw(
```

```
) ] );
our @EXPORT_OK = ( @{ $EXPORT_TAGS{'all'} } );
our @EXPORT = qw(
);
```

Массив @EXPORT должен содержать имена всех переменных и подпрограмм, которые будут автоматически импортированы вызывающим пакетом. Все они будут видны в вызывающем пакете, если мы не определим иначе.

```
use Island::Plotting::Maps; # импортировать все, что включено в массив @EXPORT
```

Ничего импортироваться не будет, если определить пустой список импорта.

```
use Island::Plotting::Maps ( ); # ничего не импортировать
```

В массиве @EXPORT_OK содержатся имена всех переменных и подпрограмм, которые могут быть импортированы, если их явно указать в списке импорта (неявно @EXPORT_OK включает в себя имена, находящиеся в массиве @EXPORT). Переменные и подпрограммы, имена которых отсутствуют в этих массивах, не смогут быть импортированы директивой use.

```
use Island::Plotting::Maps ( ); # ничего не импортировать
```

Хеш %EXPORT_TAGS позволяет присваивать имена группам переменных и функций, что дает возможность импортировать целые группы имен, не заставляя пользователя слишком много вводить с клавиатуры. Ключи хеша %EXPORT_TAGS – это имена групп, а значения – анонимные массивы имен, принадлежащих данной группе. В списке импорта директивы use имена тегов должны предваряться символом двоеточия.[1]

```
use Island::Plotting::Maps qw(:all)
```

Определив информацию для класса Exporter, создадим переменную с номером версии модуля.

```
our $VERSION = '0.01';
```

Номер версии имеет большое значение по многим причинам, и ему необходимо уделять особое внимание:

Уникальность идентификации

Номер версии будет идентифицировать конкретный выпуск конкретного модуля после его выхода на просторы Интернета.

---

1 Тег :all можно не указывать, поскольку регулярное выражение /^/ (данному регулярному выражению соответствуют любые символы с начала строки) делает то же самое. Однако многие привыкли к тегу :all, потому что смысл его более прозрачен, чем /^/.

Выбор имени файла архива

Имя файла архива с дистрибутивом по умолчанию включает в себя номер версии главного модуля. Сервер PAUSE не позволит дважды выгрузить файл с одним и тем же именем, таким образом, если программный код модуля изменился, а номер версии нет, мы рискуем испытать несколько неприятных минут.

Идентификация обновлений

Вообще увеличение номера версии свидетельствует о замещении предыдущих версий. При сравнении номеров версий числа рассматриваются как числа с плавающей запятой, по этой причине номер версии 2.10 (фактически 2.1) считается меньше номера 2.9 (2.90 – если следовать той же логике). Чтобы избежать возможных конфликтов, если в номере версии два знака после запятой, то при выпуске очередной версии номер так же должен содержать два знака после запятой.[1]

Стабильность интерфейса

Номера версий, начинающиеся нулем, считаются альфа- или бета-версиями продукта с неустоявшимся внешним интерфейсом, который может еще претерпеть некоторые изменения. Кроме того, изменение старшего номера версии (от 1.x до 2.x и т. д.) многими расценивается как признак нарушения совместимости снизу вверх между версиями.

Распознаваемость разнообразными инструментальными средствами CPAN

Инструментальные средства управления дистрибутивами в CPAN используют номер версии для поиска конкретного выпуска, а инструментальные средства установки модулей из CPAN помогают отыскать отсутствующие или устаревшие дистрибутивы.

Распознаваемость операторами языка Perl

Директиве `use` можно передавать не только имя подключаемого модуля, но и номер его версии вместе со списком импорта (или вместо него), заставив ее тем самым терпеть неудачу, если фактическая версия импортируемого модуля ниже указанной:

```
use Island::Plotting::Maps 1.10 qw{ map_debugger };
```

Вообще нумерацию версий можно начинать с 0.01, как указано в шаблоне, и затем последовательно увеличивать его с каждым новым вы-

---

[1] Документ «Perl Best Practices» рекомендует оформлять номера версий в виде строк, состоящих из трех частей, например `qv(1.2.3)`. Но такая нумерация версий не получила широкого применения из-за того, что для анализа таких номеров необходим модуль `version.pm` (который к тому же не входит в состав дистрибутива Perl). Кроме того, такое оформление не поддерживается общераспространенными инструментальными средствами и влечет за собой некоторые лексические проблемы.

пуском.[1] Теперь мы переходим от заголовочной информации к сердцу модуля. В шаблоне это место определено комментарием:

```
# Preloaded methods go here.
```

Надеюсь, вы не думали, что программа h2xs напишет программный код модуля за вас? В любом случае в этом месте должны находиться подпрограммы, которым могут предшествовать описания переменных модуля (со спецификатором my) и нескольких переменных пакета (со спецификатором our, появившимся в новых версиях Perl). Вслед за исходными текстами подпрограмм желательно вставить завершающее выражение, которое возвращает значение «истина»:

```
1;
```

благодаря которому функция require (внутри директивы use) не будет вызывать аварийное завершение программы.

Встроенная документация

Сразу же вслед за последним выражением, возвращающим значение «истина», вы найдете маркер __END__. За двумя символами подчеркивания следует слово END, за которым опять следуют два символа подчеркивания. Этот маркер стоит в начале строки и сразу же за ним расположен символ перевода строки:

```
__END__
```

Маркер сообщает Perl, что в этом месте заканчивается программный код и синтаксический анализатор Perl может завершить свою работу.[2] Вслед за маркером __END__ находится встроенная документация к модулю:

```
# Ниже находится пример документации к модулю.
# Желательно, чтобы вы отредактировали ее!

=head1 NAME

Island::Plotting::Maps - Perl extension for blah blah blah

=head1 SYNOPSIS

  use Island::Plotting::Maps;
  blah blah blah

=head1 ABSTRACT
```

[1] На страницах справочного руководства к perlmod имеется пример извлечения номера версии модуля непосредственно из системы управления версиями CVS/RCS.

[2] Данные, следующие за маркером __END__, можно прочитать с помощью дескриптора файла DATA, который представляет собой хороший способ включить в программу небольшой объем данных. Однако здесь он используется совсем с другой целью.

```
  Здесь должно находиться краткое описание модуля Island::Plotting::Maps.
  Это описание будет использовано при создании файлов PPD
  (Perl Package Description).
  Если вы не собираетесь создавать краткое описание, тогда
  отредактируйте файл Makefile.PL и удалите из него параметр ABSTRACT_FROM.

=head1 DESCRIPTION

Шаблон документации к модулю Island::Plotting::Maps. Создан программой h2xs.
Похоже, автор модуля оказался настолько небрежным, что оставил
этот шаблон без должного внимания.

Blah blah blah.

=head1 EXPORT

По умолчанию - ничего.

=head1 SEE ALSO

Здесь можно поместить ссылки на другую документацию, такую как
описание дополнительных модулей или документация к операционной
системе (например, на страницы справочного руководства ОС UNIX),
или на внешние документы, например RFC или стандарты.
Если для вашего модуля существует список рассылки, упомяните его здесь.
Если у модуля имеется собственный веб-сайт, упомяните его здесь.

=head1 AUTHOR

Ginger Grant, <ginger@island.coconet>

=head1 COPYRIGHT AND LICENSE

Copyright 2006 by Ginger Grant

Данная библиотека является свободно распространяемым программным
обеспечением.
Допускается  ее свободное распространение и/или изменение на тех же условиях,
что и сам Perl.

=cut
```

Данная документация имеет формат POD, описание которого можно найти на страницах справочного руководства к perlpod. Как и само имя Perl, название POD можно интерпретировать по-разному. Например, Perl Online Documentation (встроенная документация Perl) или Plain Old Documentation (обычная старая документация). Для большинства же из нас это просто POD.

Как говорится в шаблоне, желательно отредактировать отдельные части текста, чтобы они соответствовали вашему модулю. В частности, считается дурным тоном оставлять в документации шаблоны blah blah blah.

В процессе установки модуля информация в формате POD автоматически извлекается из него, и на ее основе будет создана документация в формате, характерном для той или иной операционной системы, например в формате страниц справочного руководства в UNIX или в формате HTML. Кроме того, с помощью команды perldoc можно будет оты-

скать документацию в формате POD в установленных сценариях и модулях и вывести ее на экран.

Самое замечательное в формате POD то, что описание программного кода может перемежаться с самим программным кодом. Все директивы POD (строки, начинающиеся с символа знака равенства) производят переключение режима работы интерпретатора из режима Perl (интерпретация программного кода) в режим POD (интерпретация текста документации), а директива `=cut` производит обратное переключение режима. Например, в модуле `Island::Plotting::Maps` у нас имеются три подпрограммы. Поэтому окончательный вариант файла, содержащий программный код, перемежаемый документацией с его описанием, может выглядеть примерно так:

```
package Island::Plotting::Maps;
[... stuff down to the $VERSION setting above ...]

=head1 NAME

Island::Plotting::Maps - Выводит карты острова

=head1 SYNOPSIS

  use Island::Plotting::Maps;
  load_map("/usr/share/map/hawaii.map");
  scale_map(20, 20);
  draw_map(*STDOUT);

=head1 DESCRIPTION

Этот модуль предназначен для рисования географических
карт. [остальное описание ]

=over

=item load_map($filename)

Данная функция [ остальное описание ].

=cut

sub load_map {
    my $filename = shift;
    [ остальная часть подпрограммы ]
}

=item scale_map($x, $y)

Данная функция [ остальное описание ].

=cut

sub scale_map {
    my ($x, $y) = (shift, shift);
    [ остальная часть подпрограммы ]
}

=item draw_map($filehandle)

Данная функция [ остальное описание ].
```

```
=cut

sub draw_map {
    my $filehandle = shift;
    [ остальная часть подпрограммы ]
}

=back

=head1 SEE ALSO

"Основы чтения географических карт для чайников",
"Первые помощники: почему они не капитаны",
Обязательно пообщайтесь с Профессором.

=head1 AUTHOR

Ginger Grant, <ginger@island.coconet>

=head1 COPYRIGHT AND LICENSE

Copyright 2006 by Ginger Grant

Данная библиотека является свободно распространяемым программным
обеспечением.
Допускается  ее свободное распространение и/или изменение на тех же условиях,
что и сам Perl.

=cut

1;
```

Как видите, описание подпрограмм находится непосредственно перед подпрограммами, и теперь, изменив саму подпрограмму, будет проще отразить эти изменения в описании к ней. (Устаревшая документация порой хуже, чем ее отсутствие, потому что при отсутствии документации пользователь вынужден будет выяснять назначение подпрограммы по ее исходным текстам.) Многие модули в CPAN следуют такому стилю оформления. Недостатком такого подхода является увеличение времени компиляции, поскольку синтаксический анализатор Perl вынужден затрачивать некоторое время на прохождение директив POD.

Неважно, каким образом оформлена документация в модуле – в конце файла или перемежается программным кодом (как было показано в предыдущих параграфах), главное, чтобы вы не забывали писать ее. Даже если модуль пишется исключительно для себя самого, через несколько месяцев, в течение которых ваш мозг был занят решением 42 000 других задач, вернувшись к этому модулю, вы будете рады увидеть описания. Документация чрезвычайно важна.

Управление дистрибутивом с помощью Makefile.PL

Разработчики Perl выбрали стандартную для операционной системы UNIX утилиту *make* в качестве основного инструмента сборки и установки Perl. Тот же самый механизм служит для установки дополни-

тельных модулей. Даже в операционной системе, отличной от UNIX, должна иметься *make*-подобная утилита. Например, в JC Windows может присутствовать утилита *dmake* или другие аналогичные программы. Команда `perl -V:make` подскажет вам точное название *make*-подобной программы. Если она выведет `make='nmake'`, то можете применять *nmake* везде, где применяется *make*. В любом случае вам необходимо будет вызывать утилиту *make* для файла *Makefile*, название которого, впрочем, также может изменяться.

Создание файла *Makefile* представляет собой задачу трудоемкую и многократно повторяющуюся. Не лучше ли выполнять сложные и повторяющиеся действия с помощью программы? Поскольку речь зашла о дополнительных модулях, то сам Perl уже установлен, и не пора ли создать с его помощью *Makefile*?

Всякий дистрибутив должен содержать файл *Makefile.PL*, который представляет собой программу на языке Perl, предназначенную для создания файла *Makefile*. После создания этого файла можно будет решить все остальные задачи посредством утилиты *make* (или аналогичной).

Программа *h2xs* генерирует заготовку программы *Makefile.PL*, к которой, скорее всего, вам даже не придется прикасаться, если в составе дистрибутива предполагается распространять единственный модуль:

```
$ cat Makefile.PL
use 5.008;
use ExtUtils::MakeMaker;
# За дополнительной информацией о том, как влиять на содержимое Makefile
# обращайтесь к lib/ExtUtils/MakeMaker.pm
WriteMakefile(
    'NAME'              => 'Island::Plotting::Maps',
    'VERSION_FROM'      => 'Maps.pm', # отыскивает $VERSION
    'PREREQ_PM'         => { }, # например, Module::Name => 1.1
     ($] >= 5.005 ? ## Добавляет ключевые слова, появившиеся в версии 5.005
       (ABSTRACT_FROM => 'Maps.pm', # извлекает краткое описание из модуля
        AUTHOR        => 'Ginger Grant <ginger@island.coconet>') : ( )),
);
```

Да, это программа на языке Perl. Подпрограмма `WriteMakefile`, которая создает *Makefile*, определена в модуле `ExtUtils::MakeMaker` (входит в состав дистрибутива Perl). Как разработчик модуля вы можете использовать эту программу для сборки и проверки модуля и для создания файла дистрибутива:

```
$ perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Island::Plotting::Maps
```

Конечный пользователь вашего дистрибутива сможет выполнить аналогичную команду у себя на компьютере. Однако у него *Makefile* наверняка будет отличаться от вашего – это зависит от того, куда выпол-

няется установка, какие локальные политики применяются и даже от того, какой компилятор языка C установлен и какие инструкции связывания должны применяться в данной архитектуре. Такой подход прекрасно зарекомендовал себя за долгие годы практики.

Процесс создания *Makefile.PL* (и соответственно получающегося *Makefile*) отличается высокой степенью гибкости. Например, имеется возможность запросить у пользователя, устанавливающего ваш модуль, имя каталога, где находятся другие библиотеки и инструментальные средства, или получить дополнительные параметры, необходимые для выполнения разнообразных действий.[1]

Параметр `PREREQ_PM` очень важен, если работоспособность модуля зависит от других модулей, которые не входят в состав стандартного дистрибутива Perl, особенно если планируется передача модуля в CPAN. Корректная работа со списком предварительно установленных модулей поможет сделать установку вашего модуля практически безболезненной, и ваши пользователи будут благодарны вам за это.

Изменение каталога установки (PREFIX=...)

Файл *Makefile*, создаваемый программой *Makefile.PL* с параметрами по умолчанию, предполагает установку модуля в системный каталог Perl, который доступен всем программам Perl через встроенную переменную списка каталогов `@INC`.

Однако на этапе тестирования модуля вы наверняка не захотите устанавливать его в общесистемный каталог, поскольку это может привести к уничтожению предыдущей установленной версии модуля и к потере работоспособности всех программ, которые используют этот модуль.

Кроме того, если вы не обладаете привилегиями системного администратора, скорее всего общесистемный каталог будет вам недоступен для записи, поскольку в противном случае любой пользователь смог бы внедрить в систему троянскую программу.[2]

К счастью, *Makefile* создает все условия для изменения каталога установки сценариев, страниц справочного руководства и библиотек. Самый простой способ изменить каталог установки состоит в том, чтобы определить значение параметра командной строки `PREFIX`:

```
$ perl Makefile.PL PREFIX=~/Testing
```

1 Старайтесь свести число запросов к минимуму. Большинству пользователей не нравится, когда им задают много вопросов, особенно если выполняется лишь обновление версии модуля. Если возможно, сохраняйте ответы на вопросы в файле с настройками модуля, чтобы в следующий раз, когда будет производиться обновление версии модуля, программа установки могла взять их как значения по умолчанию.

2 Даже если вы не являетесь системным администратором, рано или поздно вы вполне сможете получить все его привилегии.

```
Checking if your kit is complete...
Looks good
Writing Makefile for Island::Plotting::Maps
```

Сообщения, выводимые в процессе работы программы, никак не свидетельствуют об изменении каталога установки, тем не менее теперь сценарии будут устанавливаться в каталог `$PREFIX/bin`, страницы справочного руководства – в `$PREFIX/man`, а библиотеки – в `$PREFIX/lib/site_perl`. В данном случае установка будет произведена в подкаталог `Testing`, находящийся в домашнем каталоге.

Если вы занимаетесь сопровождением библиотек, используемых группой разработчиков, вы могли бы указать, например, такое значение параметра:

```
$ perl Makefile.PL PREFIX=/path/to/shared/area
```

что приведет к сборке и установке файлов в каталог общего пользования. Разумеется, при этом вы должны обладать правом записи в этот каталог. Остальным членам команды достаточно добавить путь к каталогу *bin* в переменную `PATH`, путь к каталогу *man* в переменную `MANPATH` и путь к каталогу *lib/site_perl* к списку `@INC`, как это было показано ранее.

Тривиальная команда make test

Тестирование, тестирование и еще раз тестирование. Более подробно о тестировании мы поговорим в главах 17 и 18, а здесь дадим лишь краткое введение.

Тестирование имеет особое значение. Прежде чем устанавливать свой программный код, необходимо убедиться в том, что он хотя бы компилируется. Это делается с помощью недавно созданного файла *Makefile* простой командой `make test`:

```
$ make test
cp Maps.pm blib/lib/Island/Plotting/Maps.pm
PERL_DL_NONLAZY=1 /usr/local/bin/perl "-MExtUtils::Command::MM" "-e" "
test_harness(0,
'blib/lib', 'blib/arch')" t/*.t
t/1....ok
All tests successful.
Files=1, Tests=1, 1 wallclock secs ( 0.08 cusr + 0.04 csys = 0.12 CPU)
```

Что значат все эти строки?

Сначала файл *.pm* был скопирован в каталог «испытательного полигона»: подкаталог *blib* (build library – сборка библиотеки) в текущем каталоге.[1] Затем команда *perl* исполнила сценарий *Makefile.PL*, содер-

[1] Если бы в состав дистрибутива входили файлы на языке XS или в результате более сложных действий по сборке создавались другие файлы, они также оказались бы в данном каталоге.

жащий программу испытаний – программу, в процессе работы которой исполняются тесты и в конце испытаний генерируются отчеты.[1]

В ходе выполнения программы испытаний в естественном порядке запускаются все файлы в подкаталоге *t*, которые имеют расширение *.t*. В данном случае у нас имеется всего один такой файл, созданный программой *h2xs*, содержимое которого выглядит так:

```
$ cat t/1.t
# Прежде чем выполнить команду 'make install', необходимо запустить сценарий
# командой 'make test'. После 'make install' его можно запустить
# командой 'perl 1.t'

#########################

# замените 'tests => 1' на 'tests => last_test_to_print';

use Test::More tests => 1;
BEGIN { use_ok('Island::Plotting::Maps') };

#########################

# Вставьте ниже этого комментария свой программный код теста,
# здесь подключается модуль Test::More, за дополнительной
# информацией о порядке создания тестов обращайтесь к страницам
# справочного руководства этого модуля ( perldoc Test::More ).
```

Этот сценарий представляет собой простейшую тестовую программу. Тестирование выполняется с помощью модуля `Test::More`, который будет описан в главе 17. Список импорта этого модуля интерпретируется особым образом. В данном случае объявлено, что этот файл содержит всего один «тест».

Сам тест определяется следующей строкой и представляет собой попытку подключить испытуемый модуль. Если эта операция завершается успехом, по окончании теста вы получите сообщение «OK». Данный тест мог бы закончиться неудачей, если бы модуль содержал синтаксическую ошибку или в конце модуля отсутствовало выражение, возвращающее значение «истина».

В нашем случае тест завершился благополучно, о чем свидетельствуют текст сообщения и сведения о времени, затраченном на исполнение теста.

Тривиальная команда make install

Теперь, когда точно известно, что модуль компилируется, можно попробовать его установить. Разумеется, модуль будет устанавливаться

---

[1] Команда *perl*, которая исполнила сценарий *Makefile.PL*, очень широко задействуется во всех решениях, связанных с настройками. Если на машине установлено несколько версий Perl, убедитесь, что сценарий *Makefile.PL* запускается в правильной версии. В таких случаях всегда указываются полные пути, что исключает вероятность вызвать не ту команду.

в каталог, имя которого было определено параметром PREFIX на предыдущем шаге. Этого вполне достаточно, чтобы показать, как будет протекать процесс установки у пользователя.[1] Установка запускается командой make install:

```
$ make install
Manifying blib/man3/Island::Plotting::Maps.3
Installing /home/ginger/Testing/lib/site_perl/5.8.7/Island/Plotting/Maps.pm
Installing /home/ginger/Testing/man/man3/Island::Plotting::Maps.3
Writing /home/ginger/Testing/lib/site_perl/5.8.7/darwin/auto/Island/
Plotting/Maps/.
packlist
Appending installation info to /home/ginger/Testing/lib/site_perl/5.8.7/
darwin/
perllocal.pod
```

Обратите внимание, что модуль устанавливается в каталог $PREFIX/lib/site_perl (предполагается, что ранее в параметре PREFIX был определен путь к каталогу */home/ginger/Testing*), а страница справочного руководства – в каталог $PREFIX/man (в ОС UNIX, например, в раздел 3, где находятся все описания подпрограмм). Страницы справочного руководства создаются автоматически, когда извлекаются данные в формате POD и преобразуются в формат troff -man, который понимает команда *man* ОС UNIX.[2]

Тривиальная команда make dist

Получив некоторый опыт тестирования, вы можете решить, что пора передать свой модуль друзьям и коллегам. Для этого надо создать один-единственный файл дистрибутива. Способы есть разные, но в современных версиях UNIX более широко распространено создание сжатого файла архива в формате GNU *gzip*, имеющего расширение *.tar.gz* или *.tgz*.

Для получения файла достаточно выполнить команду make dist:

```
$ make dist
rm -rf Island-Plotting-Maps-0.01
/usr/local/bin/perl "-MExtUtils::Manifest=manicopy,maniread" \
        -e "manicopy(maniread( ),'Island-Plotting-Maps-0.01', 'best');"
mkdir Island-Plotting-Maps-0.01
mkdir Island-Plotting-Maps-0.01/t
tar cvf Island-Plotting-Maps-0.01.tar Island-Plotting-Maps-0.01
```

---

[1] Если вы повторяете описываемые действия у себя, убедитесь, что пробный модуль устанавливается во временный каталог. Удалить установленные модули, как правило, довольно сложно, но закончив эксперименты, вы сможете без затей удалить тестовый каталог со всем его содержимым.

[2] В системах, отличных от UNIX, и даже в некоторых версиях UNIX это может делаться по-разному, но конечный результат будет тем же самым.

```
Island-Plotting-Maps-0.01/
Island-Plotting-Maps-0.01/Changes
Island-Plotting-Maps-0.01/Makefile.PL
Island-Plotting-Maps-0.01/MANIFEST
Island-Plotting-Maps-0.01/Maps.pm
Island-Plotting-Maps-0.01/README
Island-Plotting-Maps-0.01/t/
Island-Plotting-Maps-0.01/t/1.t
rm -rf Island-Plotting-Maps-0.01
gzip --best Island-Plotting-Maps-0.01.tar
```

Теперь у нас имеется файл с именем *Island-Plotting-Maps-0.01.tar.gz*. Номер версии, присутствующий в имени файла, был извлечен из переменной `$VERSION`.[1]

Дополнительные каталоги с библиотеками

Каталог установки библиотек определяется относительно каталога, указанного в параметре `PREFIX`. Если Джинджер в `PREFIX` укажет каталог */home/ginger/Testing*, вам придется добавить соответствующий каталог с библиотеками в список каталогов поиска. Директива `use lib`:

```
use lib '/home/ginger/Testing/lib/site_perl';
```

сделает все необходимое, чтобы позволить отыскать в этом каталоге подкаталог с требуемой версией и с учетом архитектурных особенностей операционной системы, если в этом есть необходимость (как правило, это относится к платформозависимым файлам, таким как скомпилированные модули).

Кроме того, дополнительные каталоги можно подключить с помощью параметра командной строки `-M`:

```
$ perl -M lib=/home/ginger/Testing/lib/site_perl myproggy
```

или с помощью параметра `-I`:

```
$ perl -I /home/ginger/Testing/lib/site_perl myproggy
```

или настроив соответствующим образом переменную окружения `PERL5LIB` (ниже приводится sh-подобный синтаксис):

```
$ PERL5LIB=/home/ginger/Testing/lib/site_perl; export PERL5LIB
$ ./myproggy
```

Недостатком этих методов (за исключением метода на основе `use lib`) является необходимость ввода с клавиатуры дополнительных параметров. Если некто или нечто (например, сотрудник компании или веб-сервер) пожелает запустить вашу программу, он вряд ли будет знать,

[1] Если в состав дистрибутива включается несколько модулей, в файле *Makefile.PL* необходимо определить главный модуль.

что необходимо определить дополнительную переменную среды окружения или задать дополнительные параметры командной строки, и ваша программа будет завершаться по ошибке, потому что не сможет отыскать требуемый модуль.

Старайтесь по возможности применять метод `use lib`. Остальные методы пригодны разве что для пробных запусков новых версий модулей перед заменой старых (и, возможно, для того, чтобы привести в нерабочее состояние программы, которые их используют).

Упражнения

Ответы на эти вопросы вы найдете в приложениия А в разделе «Ответы к главе 16».

Упражнение [30 мин]

Создайте дистрибутивы классов `Animal` и `Horse` из главы 12. Добавьте к ним соответствующие описания подпрограмм в формате POD. Протестируйте модуль, установите его локально и соберите файл дистрибутива. Если найдете еще немного свободного времени, распакуйте модуль в другой каталог, определите новое значение параметра `PREFIX` и установите модуль еще раз, чтобы убедиться, что в файле дистрибутива имеется все необходимое.

17

Основы тестирования

В главе 16 отмечалось, что дистрибутивы содержат функциональные возможности тестирования, доступ к которым открывает команда `make test`. Конечный пользователь получает возможность писать и запускать тесты в процессе разработки и сопровождения модуля, а также проверять работоспособность модуля в реальном окружении.

В данной книге вопросы тестирования модулей рассмотрены довольно коротко, а тех, кому эта тема интересна, отсылаем к книге Яна Лэнгворта (Ian Langworth) «Perl Testing: A Developer's Notebook», O'Reilly, 2005.

Чем больше тестов, тем лучше программный код

Зачем тестировать модуль на этапе разработки? Дело в том, что тесты позволяют выявлять допущенные огрехи на самых ранних стадиях и стимулируют разработку программ небольшими порциями (так их легче проверять), что вообще считается хорошей практикой. На первый взгляд, конечно, может показаться, что тестирование – это лишняя трата времени. Однако эта методика позволяет ускорить разработку, т. к. меньше времени уходит на отладку. Большинство затруднений устраняются еще до того, как они станут настоящими проблемами, ведь тесты выявляют именно те неисправности, которые нужно ликвидировать.

Кроме того, тесты психологически облегчают изменение программного кода, потому что показывают ошибку сразу. Мы можем уверенно отвечать руководству или коллегам на вопросы о своем программном коде, т. к. тесты позволяют увидеть, насколько он работоспособен.

Разные люди по-разному понимают тестирование. Приверженцы направления, известного как «разработка, управляемая тестами» (Test-Driven Development), утверждают, что тесты следует писать раньше, чем программный код, который должен тестироваться. При первом же запуске тест терпит неудачу. Какой толк от теста, если он не выявляет ошибку, хотя и должен? Если тест терпит неудачу, то мы пишем программный код, который обеспечит прохождение теста. Пройдя этот тест, можно создать тест для другой функции и повторить весь процесс с начала. Так и проверяется функциональность программы.

Однако на самом деле тестирование никогда не будет исчерпывающим. Наборы тестов никогда не следует уничтожать, даже после поставки модуля заказчику. Если только вы не пишете мифический модуль, «абсолютно свободный от ошибок», пользователи будут время от времени присылать сообщения об ошибках. На основе каждого из них можно создать новый набор тестов.[1] Пока вы будете устранять ошибку, остальные тесты не дадут программному коду регрессировать до менее функциональной версии, отсюда и название – *регрессивное тестирование*.

Устранив ошибки, можно подумать о выпуске новой версии. Если мы хотим расширить функциональные возможности, то сначала создаем новые тесты.[2] Поскольку существующие тесты гарантируют восходящую совместимость, можно быть уверенным, что новая версия обладает теми же функциональными возможностями, что и предыдущая, и даже немного большими.

Простейший сценарий с тестами

Прежде чем двинуться дальше, напишем несколько строк на языке Perl. Мы только что говорили о том, что тесты должны создаваться раньше, чем сам программный код, поэтому постараемся следовать своим же рекомендациям. К более подробному обсуждению мы вернемся чуть позже, а пока рассмотрим модуль `Test::More`,[3] который входит в стандартный дистрибутив Perl.

Чтобы приступить к тестированию, необходимо написать простой сценарий на языке Perl. Все тесты будут находиться в этом сценарии, поэтому мы сможем использовать все имеющиеся у нас знания о языке Perl. Подключим модуль `Test::More` и сообщим ему, сколько тестов мы собираемся запускать. Ниже добавим обращения к функциям из моду-

1 Те, кому мы посылаем отчет об ошибке, обнаруженной в их программном коде, по достоинству оценят наши усилия, если мы пошлем заодно и тест, выявляющий эту ошибку. Оценка наших усилий будет еще выше, если мы предложим и способы устранения ошибки.

2 И одновременно вносятся дополнения в документацию.

3 Существует еще один модуль – `Test::Simple`, однако большинство разработчиков предпочитают его преемника, модуль `Test::More`.

ля Test::More, подробнее о которых мы поговорим позже. А пока лишь заметим, что большинство из них сравнивают первый аргумент (полученный результат) со вторым (ожидаемым результатом).

```
#!/usr/bin/perl

use Test::More tests => 4;

ok( 1, '1 - истина' );

is( 2 + 2, 4, 'Сумма равна 4' );

is( 2 * 3, 5, 'Произведение равно 5' );

isnt( 2 ** 3, 6, "Результат не равен 6" );

like( 'Alpaca Book', qr/alpaca/i, 'Обнаружено упоминание об alpaca!' );
```

В процессе работы сценарий будет выполнять тесты и выводить результаты. Если полученное и ожидаемое значения совпадают, функции Test::More выводят строку, начинающуюся с ok. Если эти значения не совпадают, строки начинаются с not ok. Кроме этого выводится дополнительная диагностическая информация о том, что ожидал получить тот или иной тест и что он фактически получил.

```
1..4
ok 1 - 1 - истина
ok 2 - Сумма равна 4
not ok 3 - Произведение равно 5
# Failed test 'Произведение равно 5'
# in /Users/brian/Desktop/test at line 9.
#        got: '6'
#   expected: '5'
# Looks like you planned 4 tests but ran 1 extra.
# Looks like you failed 1 test of 5 run.
ok 4 - Результат не равен 6
ok 5 - Обнаружено упоминание об alpaca!
```

Далее мы покажем, как из таких результатов с помощью Perl получить удобочитаемый отчет. Функциональность модуля Test::Harness поможет избежать необходимости просматривать весь вывод, полученный при тестировании, в поисках самых важных участков.

Искусство тестирования

Хорошие тесты, кроме того, содержат небольшие примеры, иллюстрирующие документацию. Это еще один способ выразить те же самые мысли, и кому-то нравится одно, кому-то – другое.[1] Хорошие тесты придают пользователю уверенности, что программный код модуля

[1] Некоторые из модулей, применяемые нами в своей работе, гораздо проще изучать по тестовым примерам, чем по имеющейся документации в формате POD. Конечно же, любой действительно хороший пример должен повторяться в тексте документации.

(и всех его зависимостей) обладает достаточной степенью переносимости, чтобы корректно работать в его системе.

Тестирование – это целое искусство. Люди написали массу книг о том, как тестировать (и затем, похоже, забыли о них). В первую очередь важно помнить ошибки, допущенные (нами или другими) в процессе разработки, а затем убедиться с помощью тестов, что они не повторились вновь в этом проекте.

Создавая тесты, старайтесь встать на точку зрения пользователя модуля, а не разработчика. Вы знаете, как должен использоваться модуль, потому что это вы придумали его, и это у вас появилась необходимость в нем. Другие наверняка попытаются работать с модулем не совсем так, как это делаете вы. Наверное, вы и сами понимаете, что пользователи, получив такой шанс, постараются найти вашему программному код самые разнообразные применения. Вот каким должен быть ход ваших мыслей при тестировании модуля.

Во время тестирования надо как выявлять ошибки, так и подтверждать работоспособность. Проверяйте работоспособность при граничных значениях, проверяйте при обычных значениях. Проверяйте работоспособность рядом с граничными значениями – чуть ниже и чуть выше. Проверяйте функции по одной. Проверяйте функции группами. Если где-то должно быть возбуждено исключение, убедитесь, что в этот момент не возникает отрицательных побочных эффектов. Попробуйте передавать больше входных аргументов, чем надо. Попробуйте передавать меньше входных аргументов, чем надо. Попробуйте преднамеренно допустить ошибку в написании именованных параметров. Попробуйте передать слишком много или слишком мало данных. Проверьте, что происходит, когда в аргументе передается неожиданное значение, например `undef`.

Поскольку мы не можем сейчас видеть ваш модуль, попробуем протестировать функцию `sqrt` языка Perl, которая вычисляет квадратный корень числа. Для начала убедимся, что функция вычисляет правильные значения в очевидных случаях – для значений `0`, `1`, `49` и `100`. Затем проверим менее очевидный случай – выражение `sqrt(0.25)` должно возвращать значение `0.5`. И наконец, убедимся, что произведение квадратных корней числа 7 дает число, близкое к значению первоначального параметра, скажем где-то между 6,99999 и 7,00001.[1]

Теперь попробуем выразить все, что было сказано, в виде программного кода. Здесь мы задействуем те же функции, что и в предыдущем примере. Ниже приводится исходный текст сценария, проверяющий работоспособность.

```
#!/usr/bin/perl
```

[1] Не забывайте, что числа с плавающей точкой не всегда имеют абсолютную точность, – как правило, имеется некоторая погрешность. В подобных ситуациях можно воспользоваться модулем `Test::Number::Delta`.

```
use Test::More tests => 6;

is( sqrt( 0), 0, 'Корень квадратный из 0 = 0' );
is( sqrt( 1), 1, ' Корень квадратный из 1 = 1' );
is( sqrt( 49), 7, ' Корень квадратный из 49 = 7' );
is( sqrt(100), 10, ' Корень квадратный из 100 = 10' );

is( sqrt(0.25), 0.5, ' Корень квадратный из 0.25 = 0.5' );

my $product = sqrt(7) * sqrt(7);

ok( $product > 6.999 && $product < 7.001,
    "Произведение [$product] приблизительно равно 7" );
```

Мы должны убедиться, что вызов `sqrt(-1)` дает фатальную ошибку, как и вызов `sqrt(-100)`. Что произойдет в случае вызова `sqrt(&test_ sub())`, когда `&test_sub` возвращает строку `"10000"`? Что получится в результате вызова `sqrt(undef)`? А как насчет `sqrt()` или `sqrt(1,1)`? Можно ли передать функции очень большое число, например гугол: `sqrt(10**100)`? Поскольку в документации оговаривается, что функция `sqrt` работает по умолчанию с переменной `$_`, мы должны убедиться, что так оно и есть. Даже для такой простой функции, как `sqrt`, необходимо составить пару десятков тестов. Если программный код решает более сложные задачи, чем `sqrt`, для него придется написать гораздо больше тестов. Тестов никогда не бывает слишком много.

В следующем простом сценарии мы реализовали проверку дополнительных условий.[1]

```
#!/usr/bin/perl

use Test::More tests => 9;

sub test_sub { '10000' }

is( $@, '', 'изначально $@ ничего не содержит' );
{
$n = -1;
eval { sqrt($n) };
ok( $@, 'после sqrt(-1) в $@ появилось сообщение' );
}

eval { sqrt(1) };
is( $@, '', 'после sqrt(1) в $@ ничего нет' );

{
my $n = -100;
eval { sqrt($n) };
ok( $@, 'после sqrt(-100) в $@ появилось сообщение ' );
}
```

[1] Написав его, мы обнаружили, что не можем написать `sqrt(-1)`, поскольку функция `eval` не перехватывает возникающую ошибку. Вероятно, Perl перехватывает ее на этапе компиляции.

```
is( sqrt( test_sub( ) ), 100, 'Строковые значения допустимы в sqrt( )' );

eval { sqrt(undef) };
is( $@, '', 'после sqrt(undef) в $@ ничего нет' );

is( sqrt, 0, 'по умолчанию sqrt() работает с $_ (undefined)');

$_ = 100;
is( sqrt, 10, 'по умолчанию sqrt() работает с $_' );

is( sqrt( 10**100 ), 10**50, 'sqrt() справилась с числом гугол' );
```

Если вы пишете не только тесты, но и сам программный код, старайтесь думать о том, как можно будет проверить каждую его строку, чтобы весь программный код был охвачен тестами хотя бы один раз. (Можно ли протестировать ветку `else`? Можно ли протестировать ветку `elseif`?). Если вы пишете только тесты или не уверены в том, что тестами охвачен весь программный код, воспользуйтесь возможностью оценки степени охвата.[1]

Поищите другие наборы тестов. В CPAN вы сможете найти буквально десятки тысяч файлов с тестами, причем некоторые из них содержат сотни тестов. В дистрибутиве Perl также имеется несколько тысяч тестов, с помощью которых выполняется проверка корректности компиляции программного кода на разных архитектурах. Для примеров этого более чем достаточно. Михаэль Шверн (Michael Schwern) заработал звание «Perl Test Master» за то, что ему удалось полностью протестировать ядро Perl, и за то, что в сообществе постоянно слышны его призывы «тест! тест! тест!».

Тестирующая система

Обычно мы (и как разработчики, и как пользователи) запускаем тесты с помощью команды `make test`. В `Makefile` тесты запускаются под управлением модуля `Test::Harness`, который перехватывает вывод от тестов и создает отчет с результатами тестирования.

Тесты располагаются в подкаталоге `t` каталога с дистрибутивом, а каждый файл с тестами имеет расширение `.t`. Каждый файл с тестами запускается отдельно, таким образом, в случае аварийного завершения работу прекращает только текущий тестовый сценарий, а собственно тестирование продолжается.[2]

---

1 Подобные инструменты, такие как модуль `Devel::Coverage`, вы найдете в CPAN.

2 Для того чтобы прервать тестирование, достаточно «катапультироваться», выведя на стандартное устройство фразу «bail out». Это удобно, если обнаружена ошибка, способная породить целый каскад других ошибок, и при этом мы не хотим ждать, пока это произойдет.

Взаимодействие между тестовым сценарием и тестирующей системой обеспечивается при помощи вывода простых сообщений на стандартное устройство.[1] Три самых основных сообщения: о количестве тестов, об успешном прохождении тестов и о неудаче.

Каждый отдельный тестовый файл включает в себя один или более тестов. Тесты нумеруются в порядке возрастания, начиная с 1. В первую очередь тестовый сценарий должен сообщить тестирующей системе (через устройство STDOUT) ожидаемое количество тестов, которое на языке Perl выглядит как описание диапазона: 1..N. Так, если в файле содержится 17 тестов, то первой на STDOUT должна быть выведена строка:

```
1..17
```

с последующим символом перевода строки. Последнее число нужно тестирующей системе, чтобы проверить, не завершился ли раньше времени тестовый сценарий. Если тестовый сценарий лишь выполняет некоторые необязательные действия и не содержит никаких тестов, то достаточно указать строку 1..0.

Мы сами не должны беспокоиться о выводе данной строки, поскольку это делается средствами модуля Test::More. Когда этот модуль подгружается, ему сообщается, сколько тестов будет запущено, а он выводит правильно сформированную строку диапазона. Так, чтобы вывести строку 0..17, надо передать модулю Test::More число 17:

```
use Test::More tests => 17;
```

После строки диапазона номеров тестов выводятся строки с результатами, по одной на тест. Каждая строка должна начинаться со слова ok, если тест был успешно пройден, и с not ok в случае неудачи. Если бы нам приходилось писать все это вручную, тогда тест, проверяющий значение суммы чисел 1, 2 и 3, мог бы выглядеть так:

```
print +(1 + 2 = = 3 ? '', 'not '), "ok 1\n";
```

В случае неудачи мы могли бы выводить просто слово not, а слово ok выводить отдельно. Но на некоторых платформах это может вызвать определенные проблемы, что обусловлено иным способом обслуживания стандартного устройства вывода. Чтобы улучшить переносимость тестов, выводите всю строку сразу ok N или not ok N. Не забывайте о пробеле после слова not!

```
print 'not ' unless 2 * 4 == 8;
print "ok 2\n";
```

Однако во всем этом для нас нет никакой необходимости, и мы сами не должны вести счет выполненных тестов. Только представьте – вдруг

---

[1] Если вам это интересно, то точные сведения по этой теме можно найти в документации perldoc Test::Harness::Tap. Впрочем, вы найдете их там, даже если вам это неинтересно.

рано или поздно придется добавить тест где-нибудь в середине файла. Тогда нам придется вручную указать номер теста, а затем изменить номера всех последующих. Но и это еще не все. В примере, показанном выше, мы не выводим никакой полезной для нас информации в случае неудачи, таким образом, нам сложно будет разобраться с тем, что пошло не так. Разумеется, мы могли бы сделать все, что потребуется, но это очень большой объем работы! К счастью, мы можем обратиться к услугам очень удобных функций модуля Test::More.

Разработка тестов с помощью Test::More

Модуль Test::More поставляется со стандартным дистрибутивом Perl начиная с версии 5.8, а если нужен модуль для более ранней версии, то его можно взять в CPAN. Модуль работает со всеми версиями Perl начиная с версии 5.6. В следующем примере мы перепишем все тесты из предыдущего раздела, включив в них функцию ok() из модуля Test::More.

```
use Test::More tests => 4;

ok(1 + 2 == 3, '1 + 2 == 3');
ok(2 * 4 == 8, '2 * 4 == 8');
my $divide = 5 / 3;
ok(abs($divide - 1.666667) < 0.001, '5 / 3 == (примерно) 1.666667');
my $subtract = -3 + 3;
ok(($subtract eq '0' or $subtract eq '-0'), '-3 + 3 = = 0');
```

Функция ok() выводит «ok», если первый аргумент имеет значение «истина» и «not ok» в противном случае. Необязательный второй аргумент играет роль названия теста. Вслед за этой информацией функция ok() выводит символ дефиса и название теста. Таким образом, в случае неудачи тест можно будет отыскать по его названию.

```
1..4
ok 1 - 1 + 2 == 3
ok 2 - 2 * 4 == 8
ok 3 - 5 / 3 == (approx) 1.666667
ok 4 - -3 + 3 == 0
```

Теперь всю черную работу делает за нас модуль Test::More. Нам уже не надо думать о выводе информации или о нумерации тестов. Что можно сказать по поводу константы 4 в первой строке сценария? Использование подобного рода констант, наверное, оправданно, когда разработка модуля закончена и из него создается дистрибутив. Но как быть, если набор тестов постоянно расширяется? Если оставить константу в том виде, в каком она есть, тестирующая система будет постоянно сообщать о несоответствии ожидаемого и фактического количества тестов. В этом случае можно заменить числовую константу на no_plan.

```
use Test::More "no_plan";       # на время разработки

ok(1 + 2 == 3, '1 + 2 == 3');
```

```
ok(2 * 4 == 8, '2 * 4 == 8');
my $divide = 5 / 3;
ok(abs($divide - 1.666667) < 0.001, '5 / 3 == (примерно) 1.666667');
my $subtract = -3 + 3;
ok(($subtract eq '0' or $subtract eq '-0'), '-3 + 3 = = 0');
```

Теперь порядок отображения результатов работы несколько изменится – строка диапазона номеров тестов переместится в конец и будет отображать фактический диапазон номеров выполненных тестов, что может отличаться от количества, которое планировалось изначально:

```
ok 1 - 1 + 2 == 3
ok 2 - 2 * 4 == 8
ok 3 - 5 / 3 == (примерно) 1.666667
ok 4 - -3 + 3 == 0
1..4
```

Тестирующая система знает: если информация о количестве тестов в начале отсутствует, следовательно, она была перемещена в конец. Если указанное количество тестов не соответствует фактическому или в конце испытаний не было никакой информации о количестве тестов, это расценивается как ошибка выполнения серии тестов. В процессе разработки мы можем использовать описанный подход, но перед выпуском версии мы обязательно должны указать точное число тестов.

Но это еще не все! Модуль `Test::More` позволяет гораздо больше, чем просто определить значение «истина»/«ложь». Функция `is()`, которая была показана ранее, сравнивает первый аргумент (фактический результат операции) со вторым (ожидаемый результат). Необязательный третий аргумент – это, как и прежде, название теста.

```
use Test::More 'no_plan';

is(1 + 2, 3, '1 + 2 = 3');
is(2 * 4, 8, '2 * 4 = 8');
```

Обратите внимание: здесь нам удалось избавиться от операции сравнения чисел, и вместо проверки значения первого аргумента мы просто спрашиваем, равен ли фактический результат ожидаемому. В случае успешного прохождения теста данная функция не имеет никаких преимуществ перед функцией `ok()`, зато в случае неудачи она дает намного более интересные результаты.

```
use Test::More 'no_plan';

is(1 + 2, 3, '1 + 2 = 3');
is(2 * 4, 6, '2 * 4 = 6');
```

Такой сценарий выводит гораздо больше полезной информации, в которой явно говорится, что ожидала получить функция `is()` и что она получила фактически.

```
ok 1 - 1 + 2 = 3
not ok 2 - 2 * 4 = 6
```

```
#   Failed test (1.t at line 4)
#        got: '8'
#   expected: '6'
1..2
# Looks like you failed 1 tests of 2.
```

Разумеется, ошибка находится в самом тесте, но примечательно, что функция прямо сообщила о том, что было получено число 8, тогда как ожидалось число 6.[1] Это намного удобнее, чем просто знать, что «что-то пошло не так». Кроме того, существует парная функция isnt(), которая выполняет проверку на неравенство.

Теперь вернемся к третьему тесту, в котором допускается изменение фактического значения в некоторых пределах. Вместо этой функции мы можем использовать только подпрограмму cmp_ok().[2] Первый и третий аргументы функции – это операнды,[3] а второй – оператор сравнения (в виде строки!).

```
use Test::More 'no_plan';

my $divide = 5 / 3;
cmp_ok(abs($divide - 1.666667), '<' , 0.001,
    '5 / 3 должно быть (примерно) 1.666667');
```

Если в результате выполнения оператора, указанного во втором аргументе, получается значение «ложь», мы получим подробное сообщение об ошибке, в котором будут присутствовать оба значения и сам оператор сравнения.

И о последнем тесте. Хотелось бы точно видеть, что получается: 0 или -0 (в некоторых системах может возвращаться значение -0). Для этого можно воспользоваться функцией like():

```
use Test::More 'no_plan';

my $subtract = -3 + 3;
like($subtract, qr/^-?0$/, '-3 + 3 == 0');
```

Функция like() получает первый аргумент в виде строки и пытается сопоставить его со вторым аргументом. В качестве второго аргумента обычно выступают объекты регулярных выражений (в данном случае создается с помощью qr), но вполне допустимо использовать обычную строку, которая будет преобразована в объект регулярного выражения

---

1 Точнее было получено '8', а ожидалось '6'. Вы обратили внимание, что значения являются строками? Функция is() сравнивает свои аргументы как строки. Если такой способ сравнения не подходит по каким-либо причинам, то можно построить тест на базе функции ok() или cmp_ok(), о которой мы вскоре побеседуем.

2 Такого рода проверки могут быть выполнены с помощью модуля Test::Number::Delta.

3 К прискорбию поклонников обратной польской нотации.

самой функцией like(), для этого достаточно, чтобы строка выглядела как регулярное выражение:

```
like($subtract, '/^-?0$/', '-3 + 3 = = 0');
```

Представление второго аргумента в виде строки обеспечивает совместимость со старыми версиями Perl.[1] Если соответствие подтвердится, тест будет считаться пройденным. Если нет, то вместе с сообщением об ошибке будут выведены оригинальная строка и регулярное выражение. Если нужно, наоборот, убедиться в несоответствии аргументов, тогда следует использовать функцию unlike().

Тестирование объектно-ориентированных особенностей

В случае объектно-ориентированных модулей, нам необходимо убедиться, что конструктор возвращает объект определенного класса. Для этих целей существуют функции isa_ok() и can_ok():

```
use Test::More 'no_plan';

use Horse;

my $trigger = Horse->named('Триггер');
isa_ok($trigger, 'Horse');
isa_ok($trigger, 'Animal');
can_ok($trigger, $_) for qw(eat color);
```

Эти тесты получают названия по умолчанию, в результате данный сценарий выводит следующее:

```
ok 1 - The object isa Horse
ok 2 - The object isa Animal
ok 3 - Horse->can('eat')
ok 4 - Horse->can('color')
1..4
```

Здесь мы убедились, что имеем дело с лошадью, которая является разновидностью животного, а так же проверили, может ли лошадь есть и имеет ли она цвет.[2] Вслед за этим можно было бы проверить, могут ли лошади иметь уникальные клички:

```
use Test::More 'no_plan';

use Horse;

my $trigger = Horse->named('Триггер');
isa_ok($trigger, 'Horse');
my $tv_horse = Horse->named('мистер Эд');
isa_ok($tv_horse, 'Horse');
```

---

1 Форма qr// появилась только в Perl 5.005.

2 Точнее была выполнена проверка наличия методов eat и color, но мы не проверили работоспособность самих методов.

```
# Не повлияло ли создание второй лошади на кличку первой?
is($trigger->name, 'Триггер', 'правильное имя - Триггер');
is($tv_horse->name, 'мистер Эд', 'правильное имя - мистер Эд');
is(Horse->name, 'лошадь без имени');
```

Этот тест должен показать, что «лошадь без имени» и «Horse без имени» – это не одно и то же.

```
ok 1 - The object isa Horse
ok 2 - The object isa Horse
ok 3 - правильное имя - Триггер
ok 4 - правильное имя - мистер Эд
not ok 5
#     Failed test (1.t at line 56)
#          got: 'Horse без имени'
#     expected: 'лошадь без имени'
1..5
# Looks like you failed 1 test of 5.
```

Так-так! Взгляните-ка! Мы написали в тесте «лошадь без имени», а получили строку «Horse без имени». То есть ошибка в самом тесте, а не в модуле, поэтому надо исправить тест и повторить его. Если, конечно, в описании модуля фактически не указано «лошадь без имени».[1] Не надо бояться писать тесты и тестировать модули. Если где-то будет допущена ошибка, она будет обнаружена в любом случае.

Модуль Test::More позволяет даже проверить действие директивы use:

```
use Test::More 'no_plan';

BEGIN{ use_ok('Horse') }

my $trigger = Horse->named('Триггер');
isa_ok($trigger, 'Horse');
# .. остальные тесты ..
```

Различие между тестом директивы use и ее фактическим применением состоит в том, что в первом случае, когда тест терпит неудачу, работа тестового сценария прерываться не будет, хотя при этом остальные тесты также потерпят неудачу. Этот тест считается самым обычным тестом, поэтому, получив «ok», мы свободно можем положить его в копилку наших успехов для включения в еженедельный отчет.

Функция use_ok() находится в блоке BEGIN, поэтому все подпрограммы, экспортируемые модулем, будут вовремя объявлены и станут доступными для остальной части программы, как и рекомендуется в документации. Для большинства объектно-ориентированных модулей это не имеет большого значения, поскольку они, как правило, не экспортируют подпрограммы.

---

1 Получается, что тесты проверяют не только программный код, но и описание программного кода.

Списки To-Do тестов

Если тесты пишутся раньше, чем программный код, то они изначально обречены на неудачу. Можно даже добавлять тестирование новых функциональных возможностей, которое будет завершаться неудачей в период разработки. Существует ряд ситуаций, когда мы заранее знаем, что тест потерпит неудачу, и не обращаем на это внимание. Модуль Test::More предусматривает такую возможность и позволяет пометить тесты, как неготовые к использованию (TODO).

В следующем примере мы собираемся добавить метод talk() в класс Horse, но еще не сделали этого. Мы уже написали тест, так как он является частью спецификации модуля. Мы знаем, что тест будет терпеть неудачу, и это правильно, но пока мы сообщаем модулю Test::More, что неудачное завершение этого теста не будет расцениваться как ошибка.

```
use Test::More 'no_plan';

use_ok('Horse');
my $tv_horse = Horse->named('мистер Эд');

TODO: {
    local $TODO = 'лошадей еще не научили говорить';

    can_ok($tv_horse, 'talk'); # она умеет говорить!
}
is($tv_horse->name, 'мистер Эд', 'Я мистер Эд!');
```

Мы пометили программный блок меткой TODO, указав, что неудачное завершение теста – это ожидаемое явление. Внутри блока была создана локальная версия переменной $TODO, которая будет хранить причину отказа теста. Модуль Test::More при выводе результатов отмечает этот тест как тест TODO, а тестирующая система, заметив это, не будет наказывать нас за ошибку.[1]

```
ok 1 - use Horse;
not ok 2 - Horse->can('talk') # TODO лошадей еще не научили говорить
#     Failed (TODO) test (1.t at line 7)
#     Horse->can('talk') failed
ok 3 - Я мистер Эд!
1..3
```

Пропуск тестов

В некоторых случаях тесты приходится пропускать. Например, те или иные функциональные возможности тестируемого модуля могут быть доступны только при условии использования Perl определенной вер-

[1] Для создания тестов TODO необходимо иметь модуль Test::Harness версии 2.0 или выше, которая начала распространяться только в составе Perl 5.8. Однако этот модуль может быть установлен из CPAN.

сии, или в определенной операционной системе, или при наличии определенных модулей. Чтобы пропустить тест, мы должны сделать практически то же самое, что и в случае тестов TODO, но для Test::More эти два варианта сильно отличаются друг от друга.

В следующем примере для создания пропускаемого теста мы снова используем блок программного кода и отмечаем его меткой SKIP. В процессе тестирования модуль Test::More не будет исполнять этот тест, в отличие от блоков TODO, которые исполняются в любом случае. В начале блока вызывается функция skip(), которая указывает причину, по которой производится данный пропуск тестов, и их количество.

В следующем примере перед тестированием метода say_it_aloud() проверяется доступность модуля Mac::Speech. Если модуль недоступен, блок eval() вернет значение «ложь», и в результате будет вызвана функция skip().

```
SKIP: {
       skip 'Модуль Mac::Speech недоступен', 1
                unless eval { require 'Mac::Speech' };

       ok( $tv_horse->say_it_aloud( 'Я мистер Эд' );
}
```

Когда Test::More пропускает тест, он выводит сообщение ok специального вида, чтобы сохранить порядок нумерации тестов и заодно сообщить испытательной программе о том, что произошло. Позднее испытательная программа сможет отразить в своем отчете количество пропущенных тестов.

Старайтесь не использовать тесты SKIP, потому что они работают не совсем правильно. Лучше используйте для этого блоки TODO. Тесты SKIP допускается использовать только в исключительных случаях, когда они приобретают характер необязательных.

Более сложные тесты (несколько тестовых сценариев)

Изначально программа *h2xs*[1] создавала единственный тестовый сценарий с именем t/1.t[2]. Можно оставить все свои тесты в одном файле, но вообще есть смысл разбить все тесты на логически связанные группы и поместить каждую группу в отдельный файл.

---

1 Если при создании дистрибутивов вы используете другие инструментальные средства, описанные в главе 16, скорее всего, вы получите другие тестовые сценарии, возможно более сложные.

2 В Perl 5.8. В более ранних версиях создается сценарий с именем test.pl, который также работает под управлением испытательной программы, запускаемой с помощью утилиты *make*, но вывод от этого сценария перехватывается несколько иным способом.

Самый простой способ добавления дополнительных тестов заключается в создании файла `t/2.t`. Все очень просто. Нам даже не придется ничего менять ни в *Makefile.PL*, ни в испытательной программе – файл будет найден и исполнен автоматически.

Можно продолжать добавлять файлы вплоть до `9.t` и не замечать ничего необычного, но после того как будет добавлен файл с именем `10.t`, можно будет заметить, что он запускается сразу после `1.t`, перед `2.t`. Почему? Потому что тесты всегда запускаются в алфавитном порядке следования их имен. Это в общем-то неплохо, поскольку дает нам возможность в первую очередь выполнять основные тесты, а затем более экзотические всего лишь за счет изменения имен файлов.

Многие разработчики переименовывают файлы, чтобы отразить в именах назначение тестов, например `01-core.t`, `02-basic.t`, `03-advanced.t`, `04-saving.t` и т. д. Первые две цифры в имени управляют порядком исполнения сценариев, а остальная часть говорит о назначении тестов.

Как уже говорилось в главе 16, разные инструментальные средства создания дистрибутивов делают это по-разному и создают один или более тестовых сценариев. По умолчанию `Test::Harness` запускает их в том же порядке, как мы только что описали.

Кстати, Брайан написал модуль `Test::Manifest`, который позволяет обойти такую схему именования файлов. При использовании этого модуля можно создать файл `t/test_manifest` и в нем указать, какие тестовые сценарии должны исполняться и в каком порядке. Теперь файлам с тестами можно давать более осмысленные имена и не задумываться при этом о порядке их исполнения. Позднее, когда понадобится добавить новый файл с тестами, чтобы определить его местоположение в общей последовательности, достаточно будет отредактировать файл `t/test_manifest`.

Упражнения

Ответы на эти вопросы вы найдете в приложении A в разделе «Ответы к главе 17».

Упражнение [60 мин]

Создайте дистрибутив модуля, написав предварительно набор тестов к нему.

Главная цель – создать модуль `My::List::Util`, который будет экспортировать две подпрограммы: `sum()` и `shuffle()`. Подпрограмма `sum()` должна принимать список аргументов и возвращать числовую сумму их значений. Подпрограмма `shuffle()` должна принимать список аргументов, переупорядочивать его случайным образом (перетасовывать) и возвращать список с новым порядком следования элементов.

Начните разработку с подпрограммы `sum()`. Сначала напишите тесты, а потом приступайте к созданию программного кода. Когда все тесты будут проходить без ошибок, разработку можно считать законченной. Затем создайте тесты для подпрограммы `shuffle()`, после чего приступайте к ее реализации. Можно заглянуть в `perlfaq` и подсмотреть реализацию подпрограммы там.

По мере продвижения вперед обязательно вносите соответствующие изменения в документацию и в файл *MANIFEST*.

Будет еще лучше, если над этим упражнением вы будете работать с кем-нибудь в паре. Один из вас мог бы написать тесты к подпрограмме `sum()` и реализацию подпрограммы `shuffle()`, а другой наоборот – тесты к подпрограмме `shuffle()` и реализацию подпрограммы `sum()`. Обменяйтесь файлами `t/*` и посмотрите, обнаружат ли тесты какие-нибудь ошибки!

18

Дополнительные сведения о тестировании

Модуль Test::More реализует ряд простых функций тестирования общего характера, но есть еще несколько модулей Test::*, содержащих специализированные подпрограммы, предназначенные для поиска более специфичных ошибок в программном коде, благодаря чему мы можем значительно сократить объемы тестов без ухудшения их качества. Поработав с ними, вы уже от них не откажетесь.

В этой главе мы расскажем о самых популярных тестовых модулях. Они не входят в стандартный дистрибутив Perl (как, например, модуль Test::More), а если входят, то это оговаривается отдельно. Вам может показаться, что вас немного обманули, потому мы часто будем отсылать вас к документации, поставляемой с модулем, но тем самым мы мягко подталкиваем вас к выходу в мир Perl. За более подробной информацией вы можете обратиться к книге Яна Лэнгворта (Ian Langworth) «Perl Testing: A Developer's Notebook», O'Reilly, 2005, которая как раз охватывает обсуждаемую тему.

Тестирование длинных строк

В главе 17 было показано, что когда тест терпит неудачу, модуль Test::More в состоянии показать, что ожидалось получить и что было получено фактически.

```
#!/usr/bin/perl
use Test::More 'no_plan';
is( "Привет, Perl!", "Привет, perl!" );
```

В результате исполнения этого теста Test::More выведет сообщение об ошибке.

```
$ perl test.pl
not ok 1
#   Failed test (test.pl at line 5)
#          got: 'Привет, Perl!'
#     expected: 'Привет, perl!'
1..1
# Looks like you failed 1 test of 1.
```

А как быть, если сравниваемые строки имеют очень большой размер? Не очень-то хочется видеть в сообщении об ошибке строки, которые могут насчитывать сотни и тысячи символов. В подобных случаях нам достаточно увидеть начало участка, где обнаружены различия.

```
#!/usr/bin/perl

use Test::More 'no_plan';
use Test::LongString;

is_string(
    "The quick brown fox jumped over the lazy dog\n" x 10,

    "The quick brown fox jumped over the lazy dog\n" x 9 .
    "The quick brown fox jumped over the lazy camel",
);
```

Теперь сообщение об ошибке не должно содержать полные строки, а лишь тот участок, где обнаружены различия. Наш пример достаточно искусственный, однако представьте, что вы имеете дело с веб-страницей, с конфигурационным файлом или с любыми другими данными большого объема, которые нежелательно было бы целиком выводить внутри сообщения об ошибке.

```
not ok 1
#   Failed test in long_string.pl at line 6.
#          got: ..." the lazy dog\x{0a}"...
#       length: 450
#     expected: ..." the lazy camel"...
#       length: 451
#     strings begin to differ at char 447
1..1
# Looks like you failed 1 test of 1.
```

Тестирование файлов

Проверить наличие некоторого файла и определить его размер довольно просто, но чем больше объем программного кода, тем выше вероятность допустить ошибку и ввести в заблуждение программиста, который будет заниматься сопровождением модуля.

Проверить наличие файла можно было бы посредством функции `ok()` из модуля `Test::More`. Это можно сделать с помощью оператора проверки существования файла `-e`.

```
use Test::More 'no_plan';

ok( -e 'minnow.db' );
```

В принципе все работает как надо, если это действительно то, что нужно было проверить, но в этом отрывке нет ничего, что говорило бы о наших намерениях. А как быть, если, наоборот, нужно убедиться в отсутствии некоторого файла перед началом тестирования? Тест, выполняющий такую проверку, отличается от предыдущего единственным символом.

```
use Test::More 'no_plan';

ok( ! -e 'minnow.db' );
```

Чтобы сообщить о своих замыслах, здесь можно было бы добавить комментарий, но в большинстве случаев комментарии пишутся в предположении, что читающий их понимает суть происходящего. А что написать здесь? Сообщить, какая из двух ситуаций проверяется? Следует ли дополнительно указать, что тест считается пройденным в случае обнаружения файла?

```
use Test::More 'no_plan';

# проверить наличие файла
ok( ! -e 'minnow.db' );
```

Брайан написал модуль `Test::File`, имена подпрограмм которого сами поясняют суть происходящего. Если требуется сообщить, что тест будет считаться пройденным только при условии наличия файла, можно воспользоваться функцией `file_exists_ok`.

```
use Test::More 'no_plan';
use Test::File;

file_exists_ok( 'minnow.db' );
```

Сообщить, что тест будет считаться пройденным только при отсутствии файла, позволяет функция `file_not_exists_ok`.

```
use Test::More 'no_plan';
use Test::File;

file_not_exists_ok( 'minnow.db' );
```

Это достаточно простой пример, но в модуле имеется масса других функций, имена которых соответствуют той же схеме именования: первая часть имени сообщает о предмете проверки (`file_exists`), а вторая – о том, считается ли тест пройденным в случае выполнения условия (`_ok`). Такие имена функций сильно упрощают понимание смысла тестов.

```
use Test::More 'no_plan';
use Test::File;

my $file = 'minnow.db';

file_exists_ok( $file );
```

```
file_not_empty_ok( $file );
file_readable_ok( $file );
file_min_size_ok( $file, 500 );
file_mode_is( $file, 0775 );
```

Таким образом, имена функций не только сообщают о намерениях, но и повышают удобочитаемость программного кода.

Тестирование устройств STDOUT и STDERR

Одно из преимуществ функции ok() (и подобных ей) заключается в том, что они выводят информацию не в устройство STDOUT, а в дескриптор файла, являющийся дубликатом STDOUT, который создается при запуске тестового сценария. Разумеется, эти тесты возможны при условии, что STDOUT не изменяется в программе. Итак, предположим, что нужно проверить подпрограмму, выводящую некую информацию в STDOUT, например метод eat класса Horse:

```
use Test::More 'no_plan';
use_ok 'Horse';
isa_ok(my $trigger = Horse->named('Триггер'), 'Horse');

open STDOUT, ">test.out" or die "Невозможно перенаправить STDOUT! $!";
$trigger->eat("сено");
close STDOUT;

open T, "test.out" or die "Невозможно прочитать из test.out! $!";
my @contents = <T>;
close T;
is(join("", @contents), "Триггер ест сено.\n", "Триггер ест то, что надо");

END { unlink "test.out" } # убрать навоз после того, как лошадь поест
```

Обратите внимание: прежде чем выполнить тест, вывод с устройства STDOUT перенаправляется во временный файл test.out. Далее содержимое файла передается функции is(). Мы закрыли STDOUT, но функция is() по-прежнему имеет доступ к исходному устройству STDOUT, благодаря чему тестирующая система увидит результат прохождения теста – ok или not ok.

Создавая временный файл, как в этом примере, не забывайте, что текущим является каталог, где находится сценарий (даже если команда make test была запущена из другого каталога). Кроме того, старайтесь выбирать как можно более платформонезависимые имена файлов, если необходимо обеспечить переносимость сценариев.

Однако для проведения подобных тестов существует более удобный путь. Все, что было сделано выше, может сделать модуль Test::Output. Этот модуль предоставляет различные функции, которые автоматически делают все необходимое.

```
#!/usr/bin/perl
use strict;
```

```
use Test::More "noplan";
use Test::Output;

sub print_hello { print STDOUT "Добро пожаловать на борт!\n" }
sub print_error { print STDERR "В судне обнаружена пробоина!\n" }

stdout_is( \&print_hello, "Добро пожаловать на борт\n" );

stderr_like( \&print_error, qr/ship/ );
```

В первом аргументе все эти функции принимают ссылки на подпрограммы, но для нас это не проблема, поскольку данную тему мы уже обсуждали в главе 7. Если проверяться должна не подпрограмма, а лишь некоторый отрывок программного кода, достаточно будет оформить проверяемый отрывок в виде подпрограммы и использовать ее для тестирования.

```
sub test_this {
...
        }

stdout_is( \&test_this, ... );
```

Если отрывок программного кода имеет небольшой размер, можно оформить его в виде анонимной подпрограммы прямо в строке вызова функции.

```
stdout_is( sub { print "Добро пожаловать на борт" }, "Добро пожаловать
                                                      на борт" );
```

Можно даже использовать встроенные блоки программного кода, как в операторах `grep` и `map`. Обратите внимание: в этом случае, как и в операторах работы со списками `grep` и `map`, после блока программного кода запятая не ставится.

```
stdout_is { print "Добро пожаловать на борт" } "Добро пожаловать на борт";
```

Кроме модуля `Test::Output` существует еще один модуль `Test::Warn`, который позволяет выполнять аналогичные проверки с устройством `STDERR`. Правда, при работе с его функциями допускается только блочная форма записи.

```
#!/usr/bin/perl

use Test::More "noplan";
use Test::Warn;

sub add_letters { "Шкипер" + "Джиллиган" }

warning_like { add_letters( ) }, qr/non-numeric/;
```

Мы все стремимся создавать программный код, не порождающий предупреждений,[1] и можем заранее проверить, что у нас получается. Пре-

[1] При компиляции. – *Примеч. науч. ред.*

дупреждения Perl могут меняться от версии к версии, и иногда требуется узнать, где появляются новые предупреждения. В этом нам поможет модуль Test::NoWarnings, который несколько отличается от описанных выше. Он автоматически добавляет еще один тест в момент загрузки модуля, а нам остается увеличить на единицу число тестов, которое передается модулю Test::More.

```
#!/usr/bin/perl
use warnings;

use Test::More tests => 1;
use Test::NoWarnings;

my( $n, $m );
# попробовать использовать неинициализированные значения
my $sum = $n + $m;
```

В этом примере производится попытка вычислить сумму двух неинициализированных переменных. В подобных ситуациях появляется раздражающее предупреждение «use of uninitialized value» (попытка использовать неинициализированное значение) (вывод предупреждений при этом должен быть разрешен!). Что делать, если необходимо избежать появления подобных предупреждений? Модуль Test::NoWarnings покажет нам, где эти сообщения возникают, а мы сможем затем ликвидировать проблему.

```
1..1
not ok 1 - no warnings
#   Failed test 'no warnings'
#   in /usr/local/lib/perl5/5.8.7/Test/NoWarnings.pm at line 45.
# There were 2 warning(s)
#       Previous test 0 ''
#       Use of uninitialized value in addition (+) at nowarnings.pl line 6.
#
# ----------
#       Previous test 0 ''
#       Use of uninitialized value in addition (+) at nowarnings.pl line 6.
#
# Looks like you failed 1 test of 1 run.
```

Работа с ложными объектами

Иногда, чтобы проверить фрагмент системы, приходится тестировать ее целиком, хотя в работоспособности некоторых частей мы можем быть уверены на все сто процентов. Например, нет необходимости открывать массу соединений с базой данных или создавать экземпляры объектов, занимающие огромные объемы памяти, если мы тестируем только малый фрагмент кода.

Модуль Test::MockObject позволяет создавать ложные объекты. Мы передаем ему информацию о части интерфейса объекта, который хотим

тестировать, а ложный объект симулирует этот интерфейс. Как правило, метод ложного объекта не должен выполнять никаких действий, а просто сразу возвращает требуемое значение.

Например, вместо того чтобы создавать реальный объект Minnow, что в свою очередь может привести к созданию массы других объектов, мы можем создать ложный объект, который будет имитировать поведение реального объекта. После того как будет создан ложный объект, мы сохраняем его в переменной $Minnow и описываем, что должны возвращать его методы. В примере ниже мы определили, что метод engines_on должен возвращать значение «истина», а метод moored_to_dock – «ложь». В этом примере тестируется не объект корабля, а другой объект, который принимает его в качестве аргумента. Вместо того чтобы проверять работу объекта с реальным кораблем, мы подсовываем ему имитатор.

```
#!/usr/bin/perl

use Test::More 'no_plan';
use Test::MockObject;

# my $Minnow = Real::Object::Class->new( ... );
my $Minnow = Test::MockObject->new( );

$Minnow->set_true( 'engines_on' );
$Minnow->set_true( 'has_maps' );
$Minnow->set_false( 'moored_to_dock' );

ok( $Minnow->engines_on, "Двигатель работает" );
ok( ! $Minnow->moored_to_dock, "Швартовы отданы" );

my $Quartermaster = Island::Plotting->new(
    ship => $Minnow,
    # ...
    )

ok( $Quartermaster->has_maps, "Мы можем отыскать карты" );
```

Можно создать более сложные имитаторы методов, которые будут делать все, что только нам заблагорассудится. Предположим, что методы должны возвращать целые списки значений. Возможно, при этом придется симулировать соединение с базой данных и извлечение некоторых записей из таблиц. По ходу разработки нам пришлось бы несколько раз соединяться с реальной базой данных в поисках ошибки.

В следующем примере демонстрируется метод list_names ложного объекта, имитирующего доступ к базе данных, который должен возвращать список из трех имен. Нам точно известно, что объект доступа к базе данных работает правильно, и мы хотим проверить работу другого объекта (который в этом достаточно искусственном примере не показан), поэтому подменяем реальный объект ложным.

```
#!/usr/bin/perl

use Test::More 'no_plan';
```

```
use Test::MockObject;
my $db = Test::MockObject->new( );

# $db = DBI->connect( ... );
$db->mock(
        list_names => sub { qw( Джиллиган Шкипер Профессор ) }
        );

my @names = $db->list_names;

is( scalar @names, 3, 'Получено верное число результатов' );
is( $names[0], 'Джиллиган', 'Первое имя - Джиллиган' );

print "Получен список следующих имен: @names\n";
```

Тестирование документации в формате POD

Можно проверять не только программный код. Документация не менее важна, чем программный код, поскольку едва ли кто-то сможет использовать наш расчудесный модуль, если будет неизвестно, что он делает. Как уже говорилось в главе 16, в Perl применяется метод встраивания текста в формате POD в тело модуля. Этот текст может быть извлечен и выведен на экран с помощью таких инструментальных средств, как perldoc.

Как определить, не нарушили ли мы соглашения по форматированию, что может привести к невозможности извлечения документации из тела модуля. Для решения подобных проблем Брайан написал модуль Test::Pod, который просматривает исходные тексты модуля и выискивает ошибки нарушения формата POD.[1] В большинстве случаев весь программный код тестов можно взять из документации Test::Pod.

```
use Test::More;
eval "use Test::Pod 1.00";
plan skip_all => "Test::Pod 1.00 required for testing POD" if $@;
all_pod_files_ok();
```

В этом примере сначала производится попытка загрузить модуль Test::Pod с помощью строковой формы оператора eval, о которой рассказывалось в главе 2. Если попытка оканчивается неудачей, в переменную $@ записывается текст сообщения об ошибке и модулю Test::More сообщается о необходимости пропустить все оставшиеся тесты.

Если же модуль загрузился, проверка skip_all будет пройдена и выполнится функция all_pod_files_ok. Модуль отыщет все файлы POD и проверит соответствие формату. Если появится необходимость определить, какие файлы должны проверяться, мы можем сделать и это. Когда

[1] Теперь сопровождением этого модуля занимается Энди Лестер (Andy Lester). Модуль обрел такую популярность, что был включен в состав модуля Module::Starter и службы тестирования CPAN (CPAN Tester Service – CPANTS).

функция all_pod_files_ok вызывается без аргументов, она пытается отыскать все файлы в формате POD. Если ее вызвать с аргументами, то она проверит только указанные файлы.

```
use Test::More;
eval "use Test::Pod 1.00";
plan skip_all => "Test::Pod 1.00 required for testing POD" if $@;
all_pod_files_ok( 'lib/Maps.pm' );
```

Проверка соответствия документации формату POD – это еще не все, что нам доступно. Мы можем проверить наличие описаний у всех методов. Для этих целей предназначен модуль Test::Pod::Coverage. Он просматривает встроенную документацию и сообщает имена подпрограмм, описание которых не было найдено. Порядок работы с этим модулем почти такой же, как и с модулем Test::Pod.[1]

```
use Test::More;
eval "use Test::Pod::Coverage";
plan skip_all =>
    "Test::Pod::Coverage required for testing pod coverage" if $@;
plan tests => 1;
pod_coverage_ok( "Island::Plotting::Maps");
```

Оба эти модуля могут выполнить гораздо большее число проверок, чем было показано, поэтому мы рекомендуем вам обратиться к документации этих модулей за более подробной информацией по данной теме. Если для создания дистрибутива применялся модуль Module::Starter, рассмотренный в главе 16, то все эти тесты, скорее всего, уже будут созданы автоматически.

Степень покрытия тестами

Кроме всего прочего мы можем проверить степень покрытия программного кода тестами. В идеальном случае мы проверили бы нашу программу всеми возможными способами с разными входными параметрами и в разных средах окружения. Именно таким путем пошел бы Профессор, но в реальной жизни мы больше похожи на Джиллигана.[2]

Мы не будем слишком углубляться в теорию и практику проведения тестов, проверяющих степень покрытия, но вы должны понимать, что есть огромное число параметров, которые можно было бы проверить. Можно узнать, сколько операторов было выполнено в процессе тести-

[1] Потому что оба эти примера были написаны Энди Лестером.

[2] Э. Дэйкстра утверждал, что: а) тестированием в принципе нельзя охватить все возможные состояния программы; б) тестирование любой глубины может выявить наличие ошибок в обозначенных местах, но ничего не может сказать об отсутствии ошибок в программе; в) в природе не существует уже законченных и сданных программ, не содержащих ошибок, – различие состоит только в «плотности» этих скрытых ошибок. – *Примеч. науч. ред.*

рования (было охвачено) или сколько было охвачено ветвей (сколько было путей, по которым шло исполнение). Существуют и другие показатели степени покрытия, например количество охватываемых выражений. Изучение этой темы лучше начать с документации к модулю `Devel::Coverage::Tutorial`.

С модулем поставляется программа *cover*, которая выполняет большую часть необходимых действий. Для того чтобы получить характеристики конкретной программы, достаточно загрузить модуль `Devel::Cover` во время ее запуска.

```
$ perl -MDevel::Cover yourprog args
$ cover
```

В процессе работы программы модуль `Devel::Cover` создает файл, куда записывается вся собранная им информация. Программа *cover* превращает этот файл в HTML-страницы. На странице `coverage.html` собраны сводные статистические характеристики тестируемой программы. Следуя по гиперссылкам, можно получить характеристики каждого файла программы.

Тестирование дистрибутива выполняется во время проведения всех остальных тестов. Сначала надо удалить все предыдущие результаты проверки на степень покрытия, для чего программе *cover* передается параметр `-delete`. После этого запускается команда `make test`, и одновременно модулю `Devel::Cover` предоставляется возможность выполнить свою работу. Чтобы при каждом запуске тестирующей системы модуль `Devel::Cover` загружался автоматически, следует записать параметр `-MDevel::Cover` в переменную окружения `HARNESS_PERL_SWITCHES`. После этого тестирующая система при каждом вызове Perl (для каждого из тестируемых файлов) будет автоматически загружать модуль `Devel::Cover`. Запущенный сценарий сразу добавляется в базу данных проверки степени покрытия (поэтому мы удаляем базу данных перед новым запуском). Наконец, по завершении тестов снова вызывается программа *cover* и полученные сведения преобразуются в удобочитаемую форму.

```
$ cover -delete
$ HARNESS_PERL_SWITCHES=-MDevel::Cover make test
$ cover
```

Результаты исследований записываются в подкаталог *cover_db* текущего рабочего каталога. База данных с информацией о степени покрытия также помещается в подкаталог *cover_db*, а стартовая страница с результатами – в файл *cover_db/coverage.html*.

Разработка собственных модулей Test::*

Если вам потребуется провести специфические испытания, не ждите, пока кто-то напишет соответствующий модуль, а напишите собствен-

ный, обратившись для этого к модулю `Test::Builder`, который возьмет на себя решение всех проблем по интеграции с модулями `Test::Harness` и `Test::More`. Если посмотреть внимательнее, можно без особого труда заметить, что «за спиной» большинства существующих модулей `Test::*` стоит модуль `Test::Builder`.

Преимущество функций-тестов заключается в том, что они служат своего рода обертками вокруг программного кода многократного пользования. Чтобы проверить нечто, вместо последовательности операторов применяется имя функции. Это очень удобно, особенно если имя функции наглядно сообщает о смысле теста, но все гораздо сложнее, если надо выразить то же самое в виде последовательности операторов.

В главе 4 нами был написан программный код, который проверял наличие необходимых предметов экипировки у потерпевших кораблекрушение. Попробуем превратить этот программный код в модуль `Test::*`. Как и прежде, необходимые проверки будут выполняться с помощью подпрограммы `check_required_items`:

```
sub check_required_items {
    my $who = shift;
    my $items = shift;

    my @required = qw(солнечные_очки крем фляжка_с_водой накидка);
    my @missing = ( );

    for my $item (@required) {
        unless (grep $item eq $_, @$items) { # нет в списке?
            print "$who: отсутствует $item.\n";
            push @missing, $item;
        }
    }

    if (@missing) {
        print "Добавлены @missing в саквояж @$items пассажира $who.\n";
        push @$items, @missing;
    }
}
```

Нам надо преобразовать эту подпрограмму в модуль `Test::*`, который будет проверять наличие предметов экипировки (т. е. он не должен добавлять недостающие предметы) и выводить результат проверки. В своей основе все тестовые модули одинаковы. Свой модуль мы назовем `Test::Minnow::RequiredItems` и начнем со следующей заготовки:

```
package Test::Minnow::RequiredItems;
use strict;

use base qw(Exporter);
use vars qw(@EXPORT $VERSION);

use Test::Builder;

my $Test = Test::Builder->new( );
```

```
$VERSION = '0.10';
@EXPORT = qw(check_required_items_ok);

sub check_required_items_ok {
    # ....
}

1;
```

Модуль начинается с объявления имени пакета, вслед за которым следует директива, включающая режим ограничений, поскольку мы стараемся выглядеть достаточно опытными программистами (если эта трехчасовая экскурсия и обречена, то не из-за какой-нибудь ошибки в нашей программе). Затем загружается модуль Exporter и функция required_items_ok помещается в список @EXPORT, поскольку нам необходимо, чтобы имя этой функции было видимо в пространстве имен вызывающего пакета, о чем мы уже рассказывали в главе 15. Затем устанавливается значение переменной $VERSION, о которой говорилось в главе 16. Единственное, о чем мы пока умолчали, – это модуль Test::Builder. В начале тестового модуля создается новый экземпляр класса Test::Builder, ссылка на который записывается в лексическую переменную $Test, область видимости которой ограничивается рамками файла.[1]

Всеми испытаниями будет заниматься объект $Test. Мы удаляем все операторы вывода из check_required_items и операторы, изменяющие исходный список. После этого остается вставить операторы, которые сообщат испытательной программе, что все ok или not ok.

```
sub check_required_items {
    my $who = shift;
    my $items = shift;

    my @required = qw(солнечные_очки крем фляжка_с_водой накидка);
    my @missing = ( );

    for my $item (@required) {
        unless (grep $item eq $_, @$items) { # нет в списке?
            push @missing, $item;
        }
    }

    if (@missing) {
        ...
    }
    else{
        ...
    }
}
```

[1] Чем-то напоминает глобальную переменную, за исключением того, что она не является переменной пакета и не видна за пределами файла.

Теперь мы должны добавить программный код, который превратит нашу функцию в тестовую. Чтобы сообщить тестирующей системе о происходящем, необходимо вызывать методы объекта $Test. В любом случае последним должен вызываться метод $Test->ok(), чтобы возвращаемое значение этого метода стало возвращаемым значением самой функции[1]. Если будет обнаружено, что некоторые предметы экипировки отсутствуют, необходимо сообщить о неудачном прохождении теста, поэтому методу $Test->ok() передается значение «ложь», но перед этим вызывается метод $Test->diag() с текстом сообщения об ошибке.

```
sub check_required_items {
    my $who = shift;
    my $items = shift;

    my @required = qw(солнечные_очки крем фляжка_с_водой накидка);
    my @missing = ( );

    for my $item (@required) {
        unless (grep $item eq $_, @$items) { # нет в списке?
            push @missing, $item;
        }
    }

    if (@missing) {
        $Test->diag("$who: отсутствует $item.\n");
        $Test->ok(0);
    }
    else{
        $Test->ok(1);
    }
}
```

Вот и все. Можно было бы сделать гораздо больше, но нам это пока не требуется. После того как файл модуля Test::Minnow::RequiredItems будет сохранен, его сразу же можно будет использовать в тестовом сценарии.

```
use Test::More 'no_plan';
use Test::Minnow::RequiredItems;

my @gilligan = (
    Gilligan => [ qw(red_shirt hat lucky_socks water_bottle) ]
    );

check_required_items_ok( @gilligan );
```

Поскольку в саквояже Джиллигана находятся не все обязательные предметы экипировки, тест завершится неудачей. В результате будет выведено not_ok и диагностическое сообщение.

---

[1] Нам редко нужны возвращаемые значения функций, т. к. в большинстве случаев функции-тесты используются как простые подпрограммы, но ничто не мешает нам вернуть дополнительную осмысленную информацию.

```
not ok 1
1..1
# Джиллиган: отсутствует солнечные_очки крем накидка.
#     Failed test (/Users/Ginger/Desktop/package_test.pl at line 49)
# Looks like you failed 1 test of 1.
```

Итак, работа над модулем `Test::Minnow::RequiredItems` закончена, и надо бы его протестировать. Это делается при помощи модуля `Test::Builder::Tester`, и данный вопрос вам придется изучить самостоятельно.

Упражнения

Ответы на эти вопросы вы найдете в приложении A в разделе «Ответы к главе 18».

Упражнение 1 [20 мин]

Напишите документацию к модулю `My::List::Util`, который был создан в упражнении главы 17. Добавьте проверку документации, выполняемую с помощью модуля `Test::Pod`.

Упражнение 2 [20 мин]

Напишите собственный модуль `Test::My::List::Util` с единственной функцией `sum_ok`, которая должна принимать два аргумента: фактическую сумму и ожидаемую. Функция должна выводить диагностическое сообщение в случае несовпадения сумм.

```
my $sum = sum( 2, 2 );
sum_ok( $sum, 4 );
```

19

Передача модулей в CPAN

Вы можете распространять свои замечательные модули не только внутри своей организации, но и передать плоды своего труда всему сообществу Perl. Для этих целей существует специальный механизм, который называется Всемирная сеть архивов Perl (Comprehensive Perl Archive Network – CPAN), существующий уже более 10 лет и насчитывающий в своем составе более 9000 модулей.

Всемирная сеть архивов Perl

Коротко об истории появления и развитии CPAN мы рассказывали в главе 3, но тогда она рассматривалась с точки зрения использования. Теперь посмотрим на CPAN с точки зрения разработчика, стремящегося внести свой вклад.

Успех CPAN – явление далеко не случайное. Изначально проект задумывался таким образом, чтобы любой желающий имел возможность внести свой вклад максимально просто. Поэтому в составе CPAN насчитывается более 9000 модулей. Было бы здорово, если бы подобная модель была воспринята и в других языках программирования.[1,2]

---

1 Но когда речь заходит об организации аналогичных архивов для других языков программирования, первое, что пытаются сделать организаторы, – это установить массу ограничений, чего изначально избегали делать в CPAN.

2 Помимо организационных разногласий, о которых говорит автор, этому препятствует еще и серьезная объективная сложность – статическая структура скомпилированного кода более традиционных языков, и самый яркий пример тому – C/C++. Для широкого использования такой код должен был бы параметризироваться (что легко достигается в Perl), а поставка модулей в исходных кодах с возможностью модификации его под особенности применения не решает проблему: модификация сводит на нет все проделанное ранее тестирование, гарантированное поведение и т. д. – *Примеч. науч. ред.*

Не забывайте, что CPAN – это всего лишь огромное хранилище. В этом и заключается его привлекательность. Все остальные составляющие, например CPAN Search (*http://search.cpan.org/*), `CPAN.pm` и `CPANPLUS.pm`, просто используют это хранилище.

Первый шаг

Поскольку CPAN – это всего лишь огромное хранилище файлов, все, что вам придется сделать, это загрузить туда свой программный код. Для этого:

- Желательно оформить свой программный код в виде модуля
- Необходимо создать учетную запись на сервере PAUSE (Perl Authors Upload Server)

Учетная запись PAUSE – это ваш паспорт в мире CPAN. Учетная запись выдается всем желающим, для этого достаточно отправить запрос. С правилами оформления запроса можно ознакомиться на веб-странице *http://www.cpan.org/modules/04pause.html*. По ходу оформления вам будет предложено заполнить форму и предоставить о себе лишь самые основные сведения, такие как имя, адрес домашней веб-страницы, адрес электронной почты и предпочтительное имя учетной записи (имя пользователя). В настоящее время длина имени пользователя может составлять от четырех до девяти символов (имена некоторых давнишних пользователей состоят всего из 3 символов).[1] Получив учетную запись, можно подумать об оформлении своих модулей. Поскольку ваши модули, скорее всего, будут использоваться в программах вместе с модулями других авторов, необходимо придумать уникальные имена пакетам модулей, чтобы исключить возможность конфликтов имен с уже существующими модулями и не вводить в заблуждение тех, кто будет просматривать содержимое CPAN. К счастью, в списке Perl Modules (*modules@perl.org*) вы найдете массу добровольцев, давно работающих с CPAN, которые помогут вам избежать большинства возможных проблем.

Прежде чем отправить свое первое письмо администрации сервера PAUSE, попробуйте:

- Просмотреть список имеющихся модулей. Понять принцип именования. Возможно, что ваш модуль повторяет уже существующие, тогда, наверное, лучше оформить свой вклад в виде «заплаты» к существующему модулю.
- Посетить список архивов (ссылки вы найдете на *http://lists.perl.org/*) и ознакомиться с правилами общения. Это поможет вам избежать

[1] Изначально длина имени пользователя сервера PAUSE не могла превышать пяти символов, пока Рэндал не захотел получить имя `MERLYN`, в результате чего были внесены соответствующие изменения.

неприятных мгновений, когда будете читать ответ на свой вопрос. Возможно, вам удастся подыскать более грамотную формулировку для своего вопроса.

- Не забывайте, что все это держится благодаря усилиям добровольцев, которые далеко не идеальны, и все, что они делают для сообщества Perl, они делают в свое свободное время. Поэтому наберитесь терпения и будьте доброжелательны.

Подготовка дистрибутива

Как только будет улажен вопрос с именем модуля и модуль пройдет тестирование под этим именем (если в этом есть такая необходимость), подготовьте дистрибутив к передаче. Этот процесс мало чем отличается от подготовки дистрибутива для внутреннего использования, однако желательно принять во внимание следующие моменты:

- Дистрибутив должен содержать файл *README*. В CPAN этот файл автоматически извлекается из дистрибутива, после чего он становится доступен для просмотра тем, кто пожелает ознакомиться с функциональными возможностями модуля, прежде чем загрузить его.
- Создайте и протестируйте свои файлы *Makefile.PL* или *Build.PL*. Без них файлы тоже принимаются в CPAN, но те, кто скачивают модуль, выражают по этому поводу свое неудовольствие, потому что им приходится самостоятельно выяснять порядок его сборки и установки.
- Файл *MANIFEST* должен содержать точный список файлов, входящих в дистрибутив. Если вы добавили в архив какие-либо файлы, они обязательно должны быть указаны в файле *MANIFEST*. Для этого достаточно запустить команду `make manifest`, которая обновит содержимое файла в соответствии с текущим состоянием дистрибутива.[1,2]
- Дистрибутив должен иметь логичный номер версии. В файле *Makefile.PL* должно быть определено значение `VERSION` или `VERSION_FROM`. Если в состав дистрибутива входит единственный модуль (файл

---

[1] Если команда `make manifest` добавит в список слишком много файлов, можно создать файл *MANIFEST.SKIP*, где следует определить регулярные выражения Perl, отбирающие имена файлов, которые нужно игнорировать. После того как этот файл будет создан, команда `make manifest` удалит из списка *MANIFEST* все имена файлов, которые соответствуют указанным регулярным выражениям.

[2] В предыдущей сноске речь идет о «лишних» файлах из числа находящихся в рабочем каталоге на время сборки (временные файлы, промежуточные версии, результаты тестирования и т. д.). Как было отмечено, все без исключения файлы, составляющие конечный вид пакета, должны быть включены в *MANIFEST*. – *Примеч. науч. ред.*

с расширением *.pm*), лучше определить значение `VERSION_FROM`. Если дистрибутив состоит из нескольких модулей, то следует изменять значение `VERSION` в соответствии с версией главного модуля. Кроме того, при выпуске новой версии модуля номер версии должен наращиваться в лексографическом порядке.[1] (Вслед за версией 1.9 не должна следовать версия 1.10, поскольку, даже если вы и называете ее как «версия один точка десять», тем не менее для инструментальных средств, выполняющих сравнение версий, она будет версией 1.1.)

- В состав дистрибутива желательно включать тесты! Если вы этого еще не сделали, прочитайте главу 17. Ничто не придает такой уверенности[2] в программном коде, как несколько десятков тестов, выполненных на этапе установки. Если вы не предусмотрели проверку на наличие ошибок, как вы узнаете, что этих ошибок нет на самом деле?
- Попробуйте вызвать команду `make disttest`. Она соберет дистрибутив из файлов, перечисленных в *MANIFEST*, распакует его в отдельный каталог и запустит тесты, находящиеся в дистрибутиве. Если что-то не работает у вас, то и у того, кто загрузит ваш модуль из CPAN, оно тоже не будет работать.

Передача дистрибутива на сервер

Как только ваш дистрибутив будет готов к отправке на сервер, откройте веб-страницу на сервере PAUSE *http://pause.perl.org/pause/authenquery?ACTION=add_uri*. Здесь вам будет предложено ввести зарегистрированное имя пользователя PAUSE и пароль, после чего вы сможете выбрать один из вариантов отправки дистрибутива.

Закачать свой файл прямо по протоколу загрузки HTTP или передать для PAUSE адреса в Интернете, где находится файл архива. Помимо этого будет предоставлена возможность закачать дистрибутив по протоколу FTP на сайт *ftp://pause.perl.org/*.

Независимо от способа после отправки ваш дистрибутив должен появиться в списке внизу страницы. Возможно, вам придется подождать какое-то время, пока сервер PAUSE заберет файл, но, как правило, вся операция передачи выполняется достаточно быстро. Если рядом с названием дистрибутива вы не увидите имя своей учетной записи, обратите внимание администратора на этот факт.

После того как сервер PAUSE получит файл и определит, кому он принадлежит, дистрибутив будет добавлен в список предметного указателя. Затем вам должно прийти уведомление от индексатора по элек-

---

[1] Об этой особенности авторы уже говорили подробно. – *Примеч. науч. ред.*

[2] Точнее иллюзии уверенности. – *Примеч. науч. ред.*

тронной почте. После этого ваш модуль начнет свою жизнь в CPAN. Не забывайте, что буква N в аббревиатуре CPAN означает «Network» (сеть), поэтому может пройти несколько часов или дней, прежде чем ваш модуль попадет на все зеркала CPAN. Как правило, для этого требуется не более двух дней.

Если вы обнаружили какую-либо проблему или у вас что-то не получилось, можно обратиться с вопросом к администрации PAUSE, отправив электронное письмо по адресу *modules@perl.org*.

Объявление о выпуске модуля

Каким образом пользователи узнают о выходе вашего модуля? Соответствующие объявления автоматически помещаются сразу в нескольких местах, включая:

- Веб-страницу «Recent modules» (последние поступления) по адресу: *http://search.cpan.org/*.
- Раздел «new modules» (новые модули) по адресу: *http://use.perl.org/*.
- Ежедневные объявления в списке рассылки «Perl news» (новости Perl).
- Каналы IRC, связанные с тематикой Perl. Объявления о новых поступлениях автоматически посылаются по нескольким каналам сразу же после отправки дистрибутива на сервер.
- Списки доступных обновлений или неустановленных модулей в одной из командных оболочек CPAN, таких как `CPAN.pm` или `CPANPLUS.pm`.

Кроме того, имеется возможность подготовить короткое объявление и послать его в группу новостей Usenet – *comp.lang.perl.announce*. Отправьте свое объявление, и в течение одного-двух дней оно обойдет все новостные серверы по всему земному шару.

Тестирование на нескольких платформах

Служба тестирования CPAN Testers (*http://testers.cpan.org/*) проводит автоматическое тестирование почти всех модулей, передаваемых в CPAN. Добровольцы по всему миру автоматически получают новые версии модулей и проводят их тестирование на доступных им платформах. Таким образом, ваш модуль будет проверен практически на всех существующих платформах, во всех существующих операционных системах и с самыми разными версиями Perl (о существовании многих из которых вы даже не подозреваете). Результаты тестирования отправляются автору модуля и автоматически добавляются в базу данных службы. Результаты тестирования любого модуля вы найдете на веб-сайте службы или на веб-сайте CPAN Search (*http://search.cpan.org/*).

Нередко тестеры помогают устранить проблемы, проявляющиеся на платформах, к которым у вас нет доступа.

Подумайте о написании статьи или доклада

Авторы модулей очень часто участвуют в конференциях, посвященных языку Perl. В конце концов, кто еще так квалифицированно расскажет о модуле, как не его автор? Чем больше появляется людей, заинтересованных в использовании вашего модуля, тем лучше, поскольку они будут посылать вам отчеты об обнаруженных ошибках, предложения по улучшению и исправления.

Если вас немного пугает перспектива участия в конференциях или если вы не хотите ждать слишком долго, поищите местную группу пользователей Perl. Как правило, собрания в таких группах происходят часто, а объем группы не слишком велик, что позволит вам чувствовать себя уютнее. Информацию о ближайшей к вам группе пользователей вы можете поискать на веб-сайте группы Perl Mongers *http://www.pm.org/*. Если вы не найдете ничего более подходящего, начните именно с этой группы.

Упражнения

Ответы на эти вопросы вы найдете в приложении А в разделе «Ответы к главе 19».

Упражнение [время не ограничено]

Попробуйте написать модуль, который решал бы проблему зависания,[1] просматривая исходные тексты на языке Perl.[2] Этот модуль должен экспортировать функцию `will_halt`, которая получает строку и возвращает значение «истина», если строка является программным кодом на языке Perl (но не бесконечным циклом), и «ложь» – в противном случае.

---

[1] Проблему бесконечных циклов – одну из возможных причин «зависаний», но далеко не единственную. – *Примеч. науч. ред.*

[2] *http://en.wikipedia.org/wiki/Halting_problem* (*http://ru.wikipedia.org/wiki/Halting_problem*).



Ответы к упражнениям

Данное приложение содержит ответы к упражнениям из этой книги.

Ответы к главе 2

Упражнение 1

Ниже приводится один из возможных вариантов решения. Аргументы командной строки программа получает в виде массива @ARGV, поэтому мы можем взять его в качестве исходного списка. Оператор проверки файлов -s по умолчанию работает с переменной $_, а это, как известно, текущий элемент, проверяемый оператором grep. Имена всех файлов с размером менее 1000 байтов помещаются в массив @smaller_than_1000. Этот массив становится исходным списком для оператора map, который извлекает каждый элемент и возвращает его, добавив к нему пробелы в начале и символ перевода строки в конце.

```
#!/usr/bin/perl

my @smaller_than_1000 = grep { -s $_ < 1000 } @ARGV;

print map { " $_\n" } @smaller_than_1000;
```

Как правило, мы стараемся обойтись без промежуточного массива.

```
print map { " $_\n" } grep { -s < 1000 } @ARGV;
```

Упражнение 2

По умолчанию в качестве заранее определенного каталога программа использует домашний каталог. Когда подпрограмма chdir вызывается без аргумента, она выполняет переход в домашний каталог (это одна из немногих функций Perl, которая не использует переменную $_ по умолчанию).

Бесконечный цикл while продолжает работу до тех пор, пока не будет выполнено условие в операторе last, прерывающем цикл. Взглянем на условие поближе: здесь не просто проверяется значение «истина». Мы проверяем, определено ли регулярное выражение и имеет ли оно ненулевую длину. Таким образом, цикл будет прерван в случае получения undef (конец ввода) или пустой строки (которая получится в результате нажатия клавиши Enter).

После того как регулярное выражение будет прочитано, мы выполняем те же действия, что и в ответе на предыдущее упражнение. Правда, на сей раз в качестве исходного списка оператора grep выступает результат работы функции glob. Шаблон оборачивается в блок eval{} – на тот случай, если он не сможет быть скомпилирован (например, когда в его составе имеются непарные скобки).

```
#!/usr/bin/perl

chdir; # переход в домашний каталог

while( 1 )
{
    print "Введите регулярное выражение > ";
    chomp( my $regex = <STDIN> );
    last unless( defined $regex && length $regex );

    print map { "    $_\n" } grep { eval{ /$regex/ } }
            glob( ".* *" );
}
```

Ответы к главе 3

Упражнение 1

Вся хитрость этого упражнения заключается в том, чтобы поручить черную работу модулю. Это прекрасный пример работы с модулями! Модуль Cwd (cwd – это аббревиатура от английского «current working directory», что в переводе на русский язык означает «текущий рабочий каталог») автоматически импортирует функцию getcwd. Нам не надо задумываться над тем, как она работает, достаточно знать, что она справляется со своими обязанностями на большинстве основных платформ.

После того как путь к текущему каталогу будет записан в переменную $cwd, ее можно будет использовать в качестве первого аргумента метода catfile класса File::Spec. Второй аргумент метода выбирается из исходного списка оператором map и переносится в переменную $_.

```
#!/usr/bin/perl

use Cwd;
use File::Spec;

my $cwd = getcwd;
```

```
print map { "    " . File::Spec->catfile( $cwd, $_ ) . "\n" }
                glob( ".* *" );
```

Упражнение 2

Не думаю, что вы столкнетесь с серьезными проблемами при установке модуля. Но если это произойдет, можете обратиться с вопросами к Брайану, поскольку он является автором этого модуля, как и автором программы `cpan`, которую вы наверняка будете использовать для установки модуля.

После установки модуля нам остается лишь следовать примеру из документации к нему. Наша программа принимает номер ISBN в виде аргумента командной строки и создает новый объект `ISBN`, который сохраняется в виде переменной `$isbn`. Затем мы просто делаем все в соответствии с примером из документации.

```
#!/usr/bin/perl

use Business::ISBN;

my $isbn = Business::ISBN->new( $ARGV[0] );

print "ISBN: " . $isbn->as_string . "\n";
print "Код страны: " . $isbn->country_code . "\n";
print "Код издательства: " . $isbn->publisher_code . "\n";
```

Ответы к главе 4

Упражнение 1

Все они обращаются к одному и тому же элементу, за исключением второй строки, `${$ginger[2]}[1]`. Эту строку можно привести к виду `$ginger[2][1]`. Она обращается к массиву `@ginger`, а не к скалярной переменной `$ginger`.

Упражнение 2

Прежде всего создадим сам хеш:

```
my @gilligan = qw(красная_рубашка шляпа счастливые_носки фляжка_с_водой);
my @professor = qw(крем фляжка_с_водой рулетка батарейки радиоприемник);
my @skipper = qw(голубая_рубашка шляпа накидка солнечные_очки крем);
my %all = (
    "Джиллиган" => \@gilligan,
    "Шкипер" => \@skipper,
    "Профессор" => \@professor,
);
```

Затем передадим этот хеш первой подпрограмме:

```
check_items_for_all(\%all);
```

Первым аргументом подпрограмме передается ссылка на хеш, поэтому, чтобы получить ключи и соответствующие им значения, ссылку нужно разыменовать:

```
sub check_items_for_all {
    my $all = shift;
    for my $person (sort keys %$all) {
        check_required_items($person, $all->{$person});
    }
}
```

После этого вызывается оригинальная подпрограмма:

```
sub check_required_items {
    my $who = shift;
    my $items = shift;
    my @required = qw(крем солнечные_очки фляжка_с_водой накидка);
    my @missing = ( );
    for my $item (@required) {
        unless (grep $item eq $_, @$items) { # нет в списке?
            print "$who: отсутствует $item.\n";
            push @missing, $item;
        }
    }
    if (@missing) {
        print " Добавлены @missing в саквояж @$items пассажира $who.\n";
        push @$items, @missing;
    }
}
```

Ответы к главе 5

Упражнение 1

Фигурные скобки (в данном случае конструктор анонимного хеша) создают ссылку на хеш. Ссылка на хеш является скалярной величиной (как и любые другие ссылки), поэтому они не могут выступать в качестве значений хешей. Вероятно, автор этого программного кода намеревался присвоить значения ссылок скалярным переменным (`$passenger_1` и `$passenger_2`). Эту трудность можно обойти, заменив фигурные скобки круглыми.

Если вы попытаетесь запустить этот отрывок, Perl выведет диагностическое предупреждение. Если вы не увидели такого сообщения, возможно, вы забыли разрешить вывод предупреждений с помощью параметра командной строки `-w` или директивы `use warnings`. Даже если вы предпочитаете не получать предупреждения, тем не менее, в процессе отладки желательно включать эту возможность. (Сколько времени вам придется потратить для поиска аналогичных ошибок без использования предупреждений Perl? Сколько времени потребуется, чтобы включить вывод предупреждений?)

Как быть, если сообщение с предупреждением получено, но вы не можете понять в чем дело? В этом вам поможет страница справочного руководства perldiag. Разработчики Perl вынуждены были сделать тексты сообщений с предупреждениями очень краткими, потому что они компилируются в исполняемый файл perl (программа, которая исполняет программный код на языке Perl). Но на странице perldiag вы найдете полный перечень всех предупреждений, какие только могут вам встретиться, вместе с подробными описаниями причин, их породивших, потому что каждое сообщение – это определенная проблема, которую следует исправить.

Если вы слишком ленивы, чтобы за каждым разъяснениям обращаться к страницам справочного руководства, вставьте в программу директиву use diagnostics; . В этом случае при появлении каких-либо ошибок Perl будет извлекать из документации подробное описание возможной проблемы и выводить его на экран. Не оставляйте эту директиву по окончании отладки, в противном случае это приведет к снижению производительности программы независимо от того, возникают ошибки или нет.

Упражнение 2

Прежде всего необходимо сохранить общее количество байт, посланное всем компьютерам, поэтому в самом начале мы определяем переменную $all, куда записываем имя, которое будет обозначать все компьютеры. Разумеется, это должно быть такое имя, которое не будет совпадать ни с каким другим именем существующего компьютера. Использование дополнительной переменной упростит процесс написания программы и облегчит внесение изменений в будущем.

```
my $all = "**all machines**";
```

Цикл чтения файла практически не изменился по сравнению с тем, что дается в главе. Разве комментарии пропущены. Кроме того, цикл накапливает в элементе $all общее количество переданных байт.

```
my %total_bytes;
while (<>) {
    next if /^#/;
    my ($source, $destination, $bytes) = split;
    $total_bytes{$source}{$destination} += $bytes;
    $total_bytes{$source}{$all} += $bytes;
}
```

На следующем шаге выполняется сортировка списка. В результате сортировки имена узлов-отправителей упорядочиваются в порядке убывания количества отправленных байтов. Затем этот список передается внешнему циклу for. (Чтобы отказаться от использования промежуточного массива, операцию сортировки можно было бы вставить прямо в оператор for внешнего цикла.)

```
my @sources =
    sort { $total_bytes{$b}{$all} <=> $total_bytes{$a}{$all} }
    keys %total_bytes;

for my $source (@sources) {
    my @destinations =
        sort { $total_bytes{$source}{$b} <=> $total_bytes{$source}{$a} }
        keys %{ $total_bytes{$source} };
    print "$source: всего передано $total_bytes{$source}{$all} байтов\n";
    for my $destination (@destinations) {
        next if $destination eq $all;
        print "    $source => $destination:",
              " $total_bytes{$source}{$destination} байтов\n";
    }
    print "\n";
}
```

Внутри цикла for создается отсортированный список узлов-получателей (аналогично списку $sources), а затем выводится общее количество байтов, переданных текущим компьютером. В процессе работы внутреннего цикла for пропускается фиктивный элемент $all. Значение этого элемента было выведено в заголовке группы, почему тогда нельзя сразу вытолкнуть его из списка оператором shift и избежать необходимости многократной проверки? Ответ на этот вопрос вы найдете в сноске.[1] Эту программу можно несколько упростить. Выражение $total_bytes{$source} многократно задействуется при выводе данных во внутреннем цикле (и дважды во внешнем цикле). Это выражение может быть заменено скалярной переменной, инициализируемой в начале внешнего цикла:

```
for my $source (@sources) {
    my $tb = $total_bytes{$source};
    my @destinations = sort { $tb->{$b} <=> $tb->{$a} } keys %$tb;
    print "$source: всего передано $tb->{$all} байтов\n";
    my $destination (@destinations) {
        next if $destination eq $all;
        print " $source => $destination: $tb->{$destination} байтов\n";
    }
    print "\n";
}
```

Такой прием делает программный код более коротким и (возможно) более быстрым. Добавьте себе еще одно очко, если эта мысль приходила вам в голову. И еще одно, если догадались, что такой прием уменьшит ясность программного кода, и потому отказались от него.

---

[1] Нет никаких гарантий, что фиктивный элемент обязательно попадет в начало списка, даже если последний отсортирован. Если узел-отправитель отправлял данные только одному узлу-получателю, то его общий исходящий трафик будет равен трафику с единственным узлом-получателем, а в этом случае элементы могут быть отсортированы как угодно.

Ответы к главе 6

Упражнение 1

Решение этого упражнения очень похоже на решение второго упражнения из главы 5, но на этот раз мы воспользуемся модулем Storable.

```
use Storable;

my $all       = "**all machines**";
my $data_file = "total_bytes.data";

my %total_bytes;
if (-e $data_file) {
    my $data = retrieve $data_file;
    %total_bytes = %$data;
}

while (<>) {
    next if /^#/;
    my ($source, $destination, $bytes)   = split;

    $total_bytes{$source}{$destination} += $bytes;
    $total_bytes{$source}{$all}         += $bytes;
}

store \%total_bytes, $data_file;

### остальная часть программы осталась без изменений
```

В самом начале программы мы записываем в переменную имя файла, где будут храниться сведения о трафике за прошедшие дни. Затем из него извлекаются данные, но только в том случае, если файл уже существует.

После чтения данных из нового файла журнала полученная информация о трафике снова записывается в файл данных.

Если вы предпочтете записывать данные из хеша в файл в своем собственном формате, сделать это будет очень непросто. Проще говоря, чтобы решить это упражнение таким способом, нужно либо обладать необычайным талантом, либо потратить достаточно много времени, при этом почти наверняка ваши подпрограммы сериализации данных будут содержать ошибки.

Упражнение 2

Вероятно, следовало бы проверять успех операций, производимых с помощью модуля Storable. Некоторые ошибки он перехватывает сам (и завершает работу программы), но в некоторых ситуациях он может возвращать неопределенное значение. За дополнительной информацией по этой теме обращайтесь к документации модуля Storable. (Если вы добавили проверку возвращаемых значений функций store и retrieve, можете прибавить себе еще одно очко.)

Неплохо было бы создавать резервную копию файла с данными (или обоих файлов), чтобы упростить откат к предыдущему состоянию в случае каких-либо ошибок. Фактически можно было бы предусмотреть возможность создания нескольких резервных копий, например сохранить данные за истекшую неделю.

Наверное, имело бы смысл добавить возможность вывода данных, даже когда программе не передается новый файл системного журнала. В том виде, в каком программа уже существует, этого можно добиться, если передать ей пустой файл (например, /dev/null). Однако для достижения этого должен быть предусмотрен более простой путь. Кроме того, функциональность вывода и обновления файла данных можно было бы полностью отделить друг от друга.

Ответы к главе 7

Упражнение

```
sub gather_mtime_between {
    my($begin, $end) = @_;
    my @files;
    my $gatherer = sub {
        my $timestamp = (stat $_)[9];
        unless (defined $timestamp) {
            warn "Невозможно получить данные о $File::Find::name: $!,
                  пропущено\n";
            return;
        }
        push @files, $File::Find::name if
            $timestamp >= $begin and $timestamp <= $end;
    };
    my $fetcher = sub { @files };
    ($gatherer, $fetcher);
}
```

Эта подпрограмма довольно незатейлива. Основное требование заключается в том, чтобы получить корректные имена элементов. Когда функция stat вызывается внутри подпрограммы обратного вызова, она получает имя файла в переменной $_, но когда имя файла возвращается вызывающей программе (или при выводе текста предупреждения), оно извлекается из переменной $File::Find::name.

Если функция stat терпит неудачу, переменная $timestamp приобретает значение undef. (Такое возможно, когда, например, будет найдена «битая» символическая ссылка на файл.) В этом случае подпрограмма обратного вызова просто выводит предупреждение и возвращает управление раньше времени. Если вы не будете выполнять проверку на неопределенное значение, то получите предупреждение при выполнении сравнения с переменными $begin и $end.

Когда будет запущена программа, вызывающая данную подпрограмму, на экране должен появиться перечень файлов, которые были изменены с прошлого понедельника (если, конечно, вы не задали другой день недели).

Ответы к главе 8

Упражнение 1

В этом упражнении нам придется выводить данные тремя различными способами: в файл, с которым вы уже знакомы, в скаляр (для этого потребуется Perl версии 5.8) либо сразу и в файл, и в скаляр. Вся хитрость в том, чтобы объединить все каналы вывода информации в одной переменной, которая будет использоваться оператором `print`. Если в качестве дескриптора файла будет выступать переменная, мы сможем определить тип выходного устройства во время исполнения программы и соответственно реагировать на это.

```
#!/usr/bin/perl
use strict;

use IO::Tee;

my $fh;
my $scalar;

print "Укажите, куда выводить данные [Scalar/File/Tee]> ";
my $type = <STDIN>;

if( $type =~ /^s/i ) {
    open $fh, ">", \$scalar;
}
elsif( $type =~ /^f/i ) {
    open $fh, ">", "$0.out";
}
elsif( $type =~ /^t/i ) {
    open my $file_fh, ">", "$0.out";
    open my $scalar_fh, ">", \$scalar;
    $fh = IO::Tee->new( $file_fh, $scalar_fh );
}

my $date = localtime;
my $day_of_week = (localtime)[6];
print $fh <<"HERE";
Идентификатор процесса: $$
Дата: $date
День недели: $day_of_week
HERE

print STDOUT <<"HERE" if $type =~ m/^[st]/i;
содержимое Scalar:
$scalar
HERE
```

В этой программе мы просим пользователя указать, куда следует выводить данные, и ожидаем, что он введет «scalar», «file» или «tee». После того как пользователь закончит ввод, по первому введенному символу мы определяем, что он имел в виду (для большей гибкости мы не учитываем регистр символов).

Если пользователь выбрал «scalar», дескриптор открывается для вывода в скалярную переменную по ссылке. Если был выбран «file», открывается дескриптор файла. Файлу назначается имя, совпадающее с именем программы, которое извлекается из `$0`, с расширением `.out`. Если пользователь выбрал «tee», создаются два дескриптора, один для вывода в файл, другой для вывода в скалярную переменную, после чего оба дескриптора объединяются в объекте `IO::Tee`, ссылка на который записывается в переменную `$fh`. Независимо от того, какой метод выбран, в конечном счете используется единственная переменная `$fh`.

Не имеет никакого значения, откуда берутся данные для вывода, – это вопрос предпочтений. В данном случае мы выводим строку с текущей датой, которую получаем в результате вызова функции `localtime` в скалярном контексте, а затем получаем номер дня недели, обращаясь к ней как к массиву.

Кроме того, выводится идентификатор процесса (который находится в служебной переменной `$$`), это позволяет отличать друг от друга разные попытки запуска программы.

Наконец, если пользователь выберет вывод в скалярную переменную (как только в переменную, так и одновременно с файлом), содержимое скалярной переменной выводится на устройство `STDOUT`, чтобы дать возможность убедиться, что вывод произведен правильно.

Упражнение 2

```
use IO::File;
my %output_handles;
while (<>) {
    unless (/^(\S+):/) {
        warn "Имя не найдено, пропущена строка: $_";
        next;
    }
    my $name = lc $1;
    my $handle = $output_handles{$name} ||=
        IO::File->open(">$name.info") ||
            die "Невозможно создать файл $name.info: $!";
    print $handle $_;
}
```

В начале цикла из строки извлекается имя персонажа, для чего используется регулярное выражение. В случае отсутствия имени выводится предупреждение.

После того как имя будет получено, все символы в нем приводятся к нижнему регистру, чтобы информация о «МэриЭнн» и «Мэриэнн»

попадала в один и тот же файл. Кроме того, это удобно для именования файлов.

На первом проходе цикла необходимо создать дескриптор файла. Попробуем понять, как это делается. Оператор || имеет более высокий приоритет, чем оператор присваивания, вследствие этого он вычисляется в первую очередь. Если файл не может быть создан, программа завершается оператором die. Оператор ||= выполняет запись дескриптора файла в хеш, а оператор = параллельно записывает его в переменную $handle.

В следующий раз, когда из строки будет извлечено то же самое имя, оператор ||= не даст выполниться оставшейся части строки программного кода. Запомните: форма записи $gilligan ||= $anything эквивалентна $gilligan = $gilligan || $anything. Если переменная в левой части выражения содержит значение «ложь» (например, undef), она приобретает значение правой части выражения, но только если оно соответствует значению «истина» (как, например, дескриптор файла), в противном случае значение правой части выражения даже не вычисляется. Таким образом, если хеш уже содержит имя персонажа, в переменную $handle будет записано имеющееся значение, и повторно файл не создается.

Совершенно необязательно было определять имена персонажей в тексте программы, поскольку они будут извлекаться из файла. И это правильно, потому что в случае появления нового персонажа нам не придется изменять программу. Если чье-либо имя в исходном файле будет записано с ошибкой, для этого персонажа будет создан еще один файл с неправильным именем.

Упражнение 3

Сделать это можно следующим способом. Сначала необходимо просмотреть все аргументы в массиве @ARGV, отыскать элементы, которые не являются именами каталогов, и затем вывести сообщения с предупреждениями для таких элементов.

После этого мы снова просматриваем массив @ARGV и отыскиваем элементы, которые являются именами каталогов. Затем список каталогов, получившийся на выходе оператора grep, мы передаем оператору map, где каждая строка преобразуется в объект IO::Dir (пока будем считать, что появление ошибок здесь невозможно). Получившийся в результате список каталогов записывается в массив @dir_hs, который затем просматривается в цикле foreach, и каждый из его элементов передается подпрограмме print_content.

Внутри print_content не делается ничего необычного. Она просто перемещает значение первого аргумента в переменную $dh и затем использует его для просмотра и вывода содержимого каталога.

```
#!/usr/bin/perl -w
use strict;
```

```
use IO::Dir;

my @not_dirs = grep { ! -d } @ARGV;
foreach my $not_dir ( @not_dirs ) {
    print "$not_dir не является каталогом!\n";
}

my @dirs = grep { -d } @ARGV;

my @dir_hs = map { IO::Dir->new( $_ ) } grep { -d } @ARGV;

foreach my $dh ( @dir_hs ) { print_contents( $dh ) };

sub print_contents {
    my $dh = shift;

    while( my $file = $dh->read ) {
        next if( $file eq '.' or $file eq '..');
        print "$file\n";
        }
};
```

Ответы к главе 9

Упражнение 1

```
my @sorted =
    map $_->[0],
    sort { $a->[1] <=> $b->[1] }
    map [$_, -s $_],
    glob "/bin/*";
```

Применение оператора определения размера файла (-s) обходится довольно дорого. Кэшируя результаты операции, можно сэкономить некоторое время. Сколько? Ответ на этот вопрос вы найдете в комментариях к следующему упражнению.

Упражнение 2

```
use Benchmark qw(timethese);

my @files = glob "/bin/*";

timethese( -2, {
    Ordinary => q{
        my @results = sort { -s $a <=> -s $b } @files;
    },
    Schwartzian => q{
        my @sorted =
            map $_->[0],
            sort { $a->[1] <=> $b->[1] }
            map [$_, -s $_],
            @files;
    },
});
```

На ноутбуке Рэндала, где каталог /bin насчитывает 33 элемента, реализация обычного алгоритма (Ordinary) показала скорость 260 итераций в секунду, а реализация на основе преобразования Шварца (Schwartzian) – 500 итераций в секунду. Таким образом, за счет усложнения программного кода удалось получить почти двукратный прирост скорости. Для каталога /etc, насчитывающего 74 элемента, алгоритм на основе преобразования Шварца оказался почти в 3 раза быстрее. Вообще чем больше элементов необходимо отсортировать и чем дороже обходится вызов функции, тем больший выигрыш дает преобразование Шварца. Причем выигрыш совершенно не зависит от операционной системы.

В предыдущем издании этой книги мы допустили небольшую ошибку в этом отрывке программного кода, в результате чего преобразование Шварца выполнялось медленнее обычного алгоритма. Однажды Брайан, как раз когда читал лекцию по данной теме, обратил на это внимание и детально проанализировал его. С результатами его изысканий вы можете ознакомиться на веб-сайте Perl Monks: *http://www.perlmonks.com/&node_id=393128*.

Упражнение 3

```
my @dictionary_sorted =
    map $_->[0],
    sort { $a->[1] cmp $b->[1] }
    map {
        my $string = $_;
        $string =~ tr/A-ZA-Я/a-za-я/;
        $string =~ tr/a-za-я//cd;
         [ $_, $string ];
    } @input_list;
```

Внутри второго оператора map, который исполняется первым, создается копия $_. (На тот случай, когда исходные данные не должны изменяться.)

Упражнение 4

```
sub data_for_path {
    my $path = shift;
    if (-f $path or -l $path) {
        return undef;
    }
    if (-d $path) {
        my %directory;
        opendir PATH, $path or die "Невозможно открыть каталог $path: $!";
        my @names = readdir PATH;
        closedir PATH;
        for my $name (@names) {
            next if $name eq "." or $name eq "..";
            $directory{$name} = data_for_path("$path/$name");
```

```
        }
        return \%directory;
    }
    warn "$path не является файлом или каталогом\n";
    return undef;
}

sub dump_data_for_path {
    my $path = shift;
    my $data = shift;
    my $prefix = shift || "";
    print "$prefix$path";
    if (not defined $data) { # обычный файл
        print "\n";
        return;
    }
    my %directory = %$data;
    if (%directory) {
        print ", содержит:\n";
        for (sort keys %directory) {
            dump_data_for_path($_, $directory{$_}, "$prefix  ");
        }
    } else {
        print ", пустой каталог\n";
    }
}
dump_data_for_path(".", data_for_path("."));
```

Добавляя третий параметр (префикс) при обращении к подпрограмме вывода, мы указываем, что в выводе необходимо сделать отступ.

Когда подпрограмма рекурсивно вызывает сама себя, она добавляет в конец параметра `$prefix` два пробела. Почему в конец, а не в начало? Потому что изначально префикс может содержать не только пробелы, а когда пробелы добавляются в конец, мы сможем задать префикс любого вида. Например, подпрограмма может быть вызвана так:

```
dump_data_for_path(".", data_for_path("."), "> ");
```

При таком способе вызова каждая строка, выводимая на экран, будет предваряться указанным префиксом. Это позволит (в будущей гипотетической версии программы) использовать специального вида префикс для выделения каталогов, расположенных в смонтированных томах NFS (Network Filesystem – сетевая файловая система), или других особых случаев.

Ответы к главе 10

Упражнение 1

Ниже приводится один из способов. Начать надо с директив `package` и `use strict`:

```
package Oogaboogoo::date;
use strict;
```

Далее следуют описания массивов-констант, в которых хранятся названия дней недели и месяцев:

```
@day = qw(арк дип уап сен поп сеп кир);
@mon = qw(диз под бод род сип уакс лин сен кун физ нап деп);
```

Затем следует определение подпрограммы преобразования номера дня недели в название. Обратите внимание: к этой подпрограмме необходимо обращаться по имени `Oogaboogoo::date::day`:

```
sub day {
    my $num = shift @_;
    die "$num - неверный номер дня недели"
        unless $num >= 0 and $num <= 6;
    $day[$num];
}
```

Аналогично определена подпрограмма преобразования номера месяца в его название:

```
sub mon {
    my $num = shift @_;
    die "$num - неверный номер месяца"
        unless $num >= 0 and $num <= 11;
    $mon[$num];
}
```

И наконец, в конце пакета должно находиться обязательное выражение, возвращающее значение «истина»:

```
1;
```

Этот файл с именем `date.pm` должен находиться в подкаталоге `Oogaboogoo` одного из каталогов, перечисленных в списке `@INC`, например в текущем.

Упражнение 2

Ниже приводится один из способов:

```
use strict;
require 'Oogaboogoo/date.pm';
```

Далее получаем информацию о текущем времени:

```
my($sec, $min, $hour, $mday, $mon, $year, $wday) = localtime;
```

Затем с помощью описанных выше подпрограмм находим название дня недели и месяца:

```
my $day_name = Oogaboogoo::date::day($wday);
my $mon_name = Oogaboogoo::date::mon($mon);
```

По исторически сложившимся причинам номер года возвращается функцией `localtime` как смещение относительно 1900 года; это обстоятельство необходимо учесть:

```
$year += 1900;
```

Наконец, выводим строку с датой:

```
print "Сегодня $day_name, $mon_name $mday, $year.\n"
```

Ответы к главе 11

Упражнение 1

Ниже приводится один из возможных вариантов. Прежде всего необходимо определить класс `Animal` с единственным методом `speak`:

```
use strict;
{ package Animal;
    sub speak {
        my $class = shift;
        print "$class: ", $class->sound, "!\n";
    }
}
```

Затем следуют определения классов отдельных животных, каждого со своим «голосом»:

```
{ package Cow;
    our @ISA = qw(Animal);
    sub sound { "му-у-у" }
}
{ package Horse;
    our @ISA = qw(Animal);
    sub sound { "иго-го" }
}
{ package Sheep;
    our @ISA = qw(Animal);
    sub sound { "бе-е-е" }
}
```

Определение класса `Mouse` несколько отличается из-за необходимости добавить некоторое примечание:

```
{ package Mouse;
    our @ISA = qw(Animal);
    sub sound { "пи-пи-пи" }
    sub speak {
        my $class = shift;
        $class->SUPER::speak;
        print "[меня не видно, хотя и слышно!]\n";
    }
}
```

А теперь диалоговая часть программы:

```
my @barnyard = ( );
{
    print "укажите название животного (пустая строка завершает программу): ";
    chomp(my $animal = <STDIN>);
    $animal = ucfirst lc $animal; # преобразование в каноническую форму
    last unless $animal =~ /^(Cow|Horse|Sheep|Mouse)$/;
    push @barnyard, $animal;
    redo;
}

foreach my $beast (@barnyard) {
    $beast->speak;
}
```

Эта программа выполняет простую проверку с помощью шаблонов, чтобы убедиться, что пользователь не ввел название неизвестного животного, например `Alpaca`. Если этого не сделать, программа будет завершаться аварийно при попытке создать незнакомое животное. В главе 14 рассказано о методе `isa`, позволяющем проверить, является ли название, введенное пользователем, именем класса, порожденного от класса `Animal`, даже в том случае, если определение этого класса было создано после того, как была написана сама проверка.

Упражнение 2

Ниже приводится один из возможных вариантов. Прежде всего необходимо определить базовый класс `LivingCreature` с единственным методом `speak`:

```
use strict;
{ package LivingCreature;
    sub speak {
        my $class = shift;
        if (@_) {                 # есть что сказать?
            print "$class: '@_'\n";
        } else {
            print "$class: ", $class->sound, "\n";
        }
    }
}
```

Человек (класс `Person`) является одной из форм жизни, поэтому определяем его как класс наследник от `LivingCreature`:

```
{ package Person;
    our @ISA = qw(LivingCreature);
    sub sound { "ля-ля-ля" }
}
```

Далее следует определение класса Animal, представители которого могут издавать звуки, но не умеют говорить (говорят только с доктором Дулитлом):

```
{ package Animal;
    our @ISA = qw(LivingCreature);
    sub sound { die "все Животные должны издавать какой-либо звук" }
    sub speak {
        my $class = shift;
        die "животные не умеют говорить!" if @_;
        $class->SUPER::speak;
    }
}
{ package Cow;
    our @ISA = qw(Animal);
    sub sound { "му-у-у" }
}
{ package Horse;
    our @ISA = qw(Animal);
    sub sound { "иго-го" }
}
{ package Sheep;
    our @ISA = qw(Animal);
    sub sound { "бе-е-е" }
}
{ package Mouse;
    our @ISA = qw(Animal);
    sub sound { "пи-пи-пи" }
    sub speak {
        my $class = shift;
        $class->SUPER::speak;
        print "[меня не видно, хотя и слышно!]\n";
    }
}
```

Наконец, пробуем заставить человека что-нибудь сказать:

```
Person->speak;                  # просто ля-ля-ля
Person->speak("Привет, Мир!");
```

Обратите внимание: основная подпрограмма speak переместилась в класс LivingCreature, и это означает, что нам не нужно повторно определять ее в классе Person. Однако в классе Animal, прежде чем вызвать SUPER::speak, следует убедиться, что мы не заставляем животное говорить по-человечески.

Это не единственный способ добиться желаемого. Аналогичные результаты можно получить, если определить класс Person как подкласс Animal. (В этом случае класс LivingCreature можно было использовать в качестве базового не только для классов животных, но и для классов растений.) Но как в этом случае заставить говорить класс Person, если класс Animal не умеет говорить по определению? В этом случае метод

Person::speak должен был бы заниматься обработкой входного аргумента до или после (или вместо) вызова метода SUPER::speak.

Какой из этих способов лучше? Смотря какие классы будут вам необходимы и как вы будете их использовать. Если предполагается добавление дополнительных функциональных возможностей в класс Animal, которые потребуются и в классе Person, то есть смысл породить класс Person от класса Animal. Если же эти два класса отличаются друг от друга и их объединяют лишь признаки, характерные для всех форм жизни (LivingCreature), то лучше отказаться от лишнего уровня наследования. Способность создавать самые оптимальные структуры наследования очень важна для программиста ООП.

Нередко бывает, что, начав разработку программы одним способом, программист понимает, что необходимо провести рефакторинг и перейти на другой способ реализации. Это довольно характерно для ООП. Поэтому особую значимость приобретает этап тестирования, который помогает убедиться, что в результате внесенных структурных изменений программный код не потерял свою функциональность.

Ответы к главе 12

Упражнение

В первую очередь необходимо определить пакет Animal:

```
use strict;
{ package Animal;
    use Carp qw(croak);
```

и конструктор класса:

```
## конструкторы
sub named {
    ref(my $class = shift) and croak " требуется имя класса";
    my $name = shift;
    my $self = { Name => $name, Color => $class->default_color };
    bless $self, $class;
}
```

Далее следуют определения виртуальных методов – методов, которые должны перекрываться в дочерних классах. Язык Perl не требует объявлять виртуальные методы, но они прекрасно справляются с ролью элементов документации.

```
## заглушки (должны быть перекрыты)
sub default_color { "коричневый" }
sub sound { croak "подкласс должен объявлять метод sound" }
```

Далее следуют методы, которые могут работать как со ссылками на экземпляры, так и с именами классов:

```
## методы классов/экземпляров
```

```
sub speak {
    my $either = shift;
    print $either->name, ": ", $either->sound, "\n";
}
sub name {
    my $either = shift;
    ref $either
        ? $either->{Name}
        : "$either без имени";
}
sub color {
    my $either = shift;
    ref $either
        ? $either->{Color}
        : $either->default_color;
}
```

Наконец, методы, которые могут работать только со ссылками на экземпляры:

```
## методы экземпляров
sub set_name {
    ref(my $self = shift) or croak "требуется ссылка на экземпляр класса";
    $self->{Name} = shift;
}
sub set_color {
    ref(my $self = shift) or croak "требуется ссылка на экземпляр класса";
    $self->{Color} = shift;
}
}
```

Теперь, когда у нас имеется абстрактный базовый класс, можно приступить к описанию классов отдельных видов животных:

```
{ package Horse;
    our @ISA = qw(Animal);
    sub sound { "иго-го" }
}
{ package Sheep;
    our @ISA = qw(Animal);
    sub color { "белый" }  # перекрыть цвет по умолчанию
    sub sound { "бе-е-е" } # никакого "Молчания ягнят"
}
```

В заключение несколько строк программного кода, который проверит наши классы:

```
my $tv_horse = Horse->named("мр. Эд");
$tv_horse->set_name("мистер Эд");
$tv_horse->set_color("серый");
print $tv_horse->name, ", цвет ", $tv_horse->color, "\n";
print Sheep->name, " окрашен в ", Sheep->color, " цвет, издает звук ", Sheep-
>sound, "\n";
```

Ответы к главе 13

Упражнение

Начнем с определения класса:

```
{ package RaceHorse;
    our @ISA = qw(Horse);
```

Затем с помощью простой функции dbmopen свяжем хеш %STANDINGS с хранилищем:

```
dbmopen (our %STANDINGS, "standings", 0666)
    or die "Невозможно получить доступ к хранилищу: $!";
```

При создании экземпляра RaceHorse необходимо извлечь его показатели из хранилища или создать новую запись с нулевыми показателями:

```
sub named { # метод класса
    my $self = shift->SUPER::named(@_);
    my $name = $self->name;
    my @standings = split ' ', $STANDINGS{$name} || "0 0 0 0";
    @$self{qw(первых вторых третьих ни_одного)} = @standings;
    $self;
}
```

При уничтожении экземпляра его показатели необходимо сохранить:

```
sub DESTROY { # метод экземпляра, вызывается автоматически
    my $self = shift;
    $STANDINGS{$self->name} = "@$self{qw(первых вторых третьих ни_одного)}";
    $self->SUPER::DESTROY;
}
```

Завершают описание методы экземпляра:

```
## методы экземпляров:
sub won { shift->{wins}++; }
sub placed { shift->{places}++; }
sub showed { shift->{shows}++; }
sub lost { shift->{losses}++; }
sub standings {
    my $self = shift;
    join ", ", map "$self->{$_} $_", qw(первых вторых третьих ни_одного);
}
```

Ответы к главе 14

Упражнение 1

Решить эту проблему можно множеством способов. Мы решили создать пакет MyDate в одном файле со сценарием, его использующим. Область видимости директивы package MyDate ограничивается программным блоком. В самом сценарии не следует использовать директиву use

MyDate, поскольку Perl не сможет найти нужный файл. Кроме того, мы должны будем вызвать подпрограмму import, чтобы перенести имена подпрограмм в пространство имен пакета main.

Чтобы подпрограмма AUTOLOAD могла работать только с нужными подпрограммами, мы определили хеш %Allowed_methods, в котором будут храниться имена допустимых методов. В качестве значений ключей хеша выступают индексы в массиве, возвращаемом функцией localtime. Есть еще одна проблема – функция localtime ведет нумерацию месяцев и лет, начиная с нуля. Поэтому в массиве @Offsets мы сохраняем смещения, которые необходимо добавить к соответствующим элементам в списке localtime. На первый взгляд мы нерационально распорядились памятью, выделив массив из 9 элементов для хранения всего двух значений, однако такой прием помог обойти два специальных случая.

Далее нам нужен метод new (или любой другой конструктор), который создавал и возвращал бы объект. В данном примере внутренняя структура объекта не имеет большого значения. Мы просто будем создавать анонимный хеш, связанный с текущим пакетом (имя которого должно передаваться в первом аргументе, поэтому $_[0]). Кроме того, нам потребуется определить метод DESTROY, который автоматически вызывается при уничтожении объекта. Если этого не сделать, наш метод AUTOLOAD будет выводить предупреждение на неопределенный метод, когда будет пытаться обработать свой собственный метод DESTROY (закомментируйте объявление метода DESTROY и посмотрите, что произойдет).

Внутри подпрограммы AUTOLOAD имя метода запоминается в переменной $method, поэтому мы можем изменять его. Прежде всего надо удалить имя пакета так, чтобы осталось одно имя метода. Имя метода, как известно, находится в полном имени вслед за последней парой двоеточий, поэтому с помощью оператора подстановки мы удаляем все, что находится перед этим. Получив имя метода, мы пытаемся отыскать ключ с таким именем в хеше %Allowed_methods. Если имя метода не будет найдено, выводится сообщение об ошибке. Попробуйте обратиться к неизвестному методу. Для какой строки Perl выведет сообщение?

Если имя метода будет найдено в %Allowed_methods, мы извлекаем значение ключа, которое далее будет использоваться в качестве индекса в списке localtime. Это значение запоминается в переменной $slice_index, после чего эта переменная используется для извлечения данных из списка localtime и из массива @Offset. Мы складываем оба полученных значения и возвращаем в виде результата.

Объяснение получилось довольно длинным, похоже, что нас ждет много работы, но сколько придется потрудиться, чтобы создать новые методы? Несколько часов? Несколько минут? А здесь нам достаточно добавить несколько имен в хеш %Allowed_methods, а все остальное уже работает.

```
#!/usr/bin/perl -w
use strict;
```

```
{
    package MyDate;
    use vars qw($AUTOLOAD);

    use Carp;

    my %Allowed_methods = qw( date 3 month 4 year 5 );
    my @Offsets         = qw(0 0 0 0 1 1900 0 0 0);

    sub new { bless { }, $_[0] }
    sub DESTROY {}
    sub AUTOLOAD {
        my $method = $AUTOLOAD;
        $method =~ s/.*:://;

        unless( exists $Allowed_methods{ $method } ) {
                carp "Неизвестный метод: $AUTOLOAD";
                return;
                }

        my $slice_index = $Allowed_methods{ $method };

        return (localtime)[$slice_index] + $Offsets[$slice_index];
        }
}

    MyDate->import; # мы ее не использовали
    my $date = MyDate->new( );

    print "День: " . $date->date . "\n";
    print "Месяц: " . $date->month . "\n";
    print "Год: " . $date->year . "\n";
```

Упражнение 2

Этот сценарий выглядит точно так же, как и в ответе к предыдущему упражнению, добавилась лишь подпрограмма `UNIVERSAL::debug`. В конце сценария мы добавили вызов подпрограммы `debug` для объекта `$date`. Никаких других изменений в модуле `MyDate` не было.

```
MyDate->import; # мы ее не использовали
my $date = MyDate->new( );

sub UNIVERSAL::debug {
    my $self = shift;
    print '[' . localtime . '] ' . join '|', @_
    }

print "День: " . $date->date . "\n";
print "Месяц: " . $date->month . "\n";
print "Год: " . $date->year . "\n";

$date->debug( "Я закончил" );
```

Как это согласуется с механизмом `AUTOLOAD`? Не забывайте, что прежде чем обратиться к методу `AUTOLOAD`, Perl сначала попытается отыскать требуемый метод в дереве наследования `@ISA`, а затем в классе `UNIVERSAL`.

В данном случае Perl найдет метод `UNIVERSAL::debug` прежде, чем очередь дойдет до вызова метода `AUTOLOAD`.

Ответы к главе 15

Упражнение 1

Модуль `Oogaboogoo/date.pm` выглядит так:

```
package Oogaboogoo::date;
use strict;
use Exporter;
our @ISA = qw(Exporter);
our @EXPORT = qw(day mon);

@day = qw(арк дип уап сен поп сеп кир);
@mon = qw(диз под бод род сип уакс лин сен кун физ нап деп);

sub day {
    my $num = shift @_;
    die "$num - неверный номер дня недели"
        unless $num >= 0 and $num <= 6;
    $day[$num];
}

sub mon {
    my $num = shift @_;
    die "$num - неверный номер месяца"
        unless $num >= 0 and $num <= 11;
    $mon[$num];
}
1;
```

А вот сама программа:

```
use strict;
use Oogaboogoo::date qw(day mon);

my($sec, $min, $hour, $mday, $mon, $year, $wday) = localtime;
my $day_name = day($wday);
my $mon_name = mon($mon);
$year += 1900;
print "Сегодня $day_name, $mon_name $mday, $year.\n";
```

Упражнение 2

По сути ответ на это упражнение ничем не отличается от ответа на предыдущее упражнение. Нам нужно лишь добавить программный код, который определит тег `all`.

```
our @EXPORT = qw(day mon);
our %EXPORT_TAGS = ( all => \@EXPORT );
```

Все, что вставляется в `@EXPORT_TAGS`, должно быть доступно через `@EXPORT` или `@EXPORT_OK`. При описании тега `all` мы использовали ссылку на массив `@EXPORT`. Можно определить список и по-другому, чтобы эти два массива никак не ссылались друг на друга.

```
our @EXPORT = qw(day mon);
our %EXPORT_TAGS = ( all => [ @EXPORT ] );
```

Осталось только изменить программу из предыдущего упражнения так, чтобы директива `use` импортировала имена подпрограмм с помощью тега `all`.

Таким образом, измененный вариант основной программы должен начинаться со строк:

```
use strict;
use Oogaboogoo::date qw(:all);
```

Ответы к главе 16

Упражнение

В данном случае для решения программный код писать не надо. Мы возьмем его из главы 12 и на его основе создадим дистрибутив. Основная работа, которую придется выполнить, — это редактирование разных файлов.

Однако даже создать дистрибутив можно несколькими способами. Как только будет создана заготовка дистрибутива с помощью наиболее удобного для вас инструмента, вы можете попробовать разделить классы и вынести их в отдельные файлы модулей. У вас могут получиться файлы `Animal.pm`, `Horse.pm` и т. д. Поместите их в каталог с оригинальным файлом модуля `.pm`, который был создан с помощью инструмента создания модулей. Вам также потребуется изменить файл *Makefile.PL* или *Build.PL*, чтобы добавить в него новые файлы модулей. Для этого просто следуйте примерам, которые уже имеются в файлах. И наконец, проверьте содержимое файла *MANIFEST* и убедитесь, что в нем содержится полный список всех файлов.

Закончив приготовления, запустите *Makefile.PL*. Это надо проделывать при каждом изменении данного файла. Если изменяются файлы модуля, придется также запустить команду *make*, хотя это и так произойдет, когда будет запущена команда `make test` или `make dist`.

Когда все будет готово, запустите команду `make dist`. В текущем каталоге появится новый файл архива. Если вам потребуется получить архивный файл в формате ZIP, запустите команду `make zipdist`. Переместите архив в другой каталог и распакуйте его. Когда вы запустите *Makefile.PL*, вы не должны получить никаких сообщений об ошибках при условии, что все было сделано правильно. Если будут выведены

предупреждения, ликвидируйте проблему (скорее всего, дело будет в нехватке нескольких файлов) и повторите попытку.

Ответы к главе 17

Упражнение

Начнем с тестового файла. Сначала напишем тесты (которые нельзя будет запустить, пока не будет написан сам модуль) и по мере их написания будем продумывать интерфейс модуля.

Для начала в блоке `BEGIN` проверим возможность подключения модуля с помощью директивы use `My::List::Util`. Очевидно, что на данном этапе этот тест будет терпеть неудачу, поскольку сам модуль еще отсутствует. Но об этом мы будем беспокоиться потом.

Затем мы проверим наличие подпрограммы `sum`. После того как будет написана заготовка модуля `My::List::Util`, тест `use_ok` будет проходить без ошибок, а проверка наличия подпрограммы `sum` будет терпеть неудачу. В этом и заключается принцип разработки, управляемой тестами. Сначала вы определяете, что хотите получить, убеждаетесь, что тест не проходит, а потом пишете код, который заставит тест выполниться без ошибок. Обеспечить прохождение тестов гораздо проще, если не задумываться, почему не проходят тесты, которые и не должны проходить.

После этого мы проверим работоспособность подпрограммы `sum`. Подсчитаем несколько сумм, передавая разное число входных аргументов с разными значениями. Пока не будем останавливаться на каждом частном случае (мы всегда сможем сделать это позднее), но нам нужно создать несколько тестов, которые проверяли бы подпрограммы разными способами. Кроме обычных случаев суммирования мы добавим вариант, когда один из аргументов не является числом, и один вариант, когда оба аргумента не являются числами.

Затем перейдем к подпрограмме `shuffle`. Сначала проверим ее наличие, затем в качестве отправной точки создадим переменную `$array` и сразу же скопируем ее в переменную `$shuffled`; это даст нам возможность не беспокоиться по поводу сохранности оригинала. Еще до того как мы приступили к разработке программного кода, мы решили, что массив лучше передавать по ссылке; это позволит подпрограмме производить перестановку в самом массиве, а не в его копии.

Затем надо проверить результат. Проверка будет очень простой. Мы сравним оригинальный массив с его измененной копией, для чего воспользуемся функцией `cmp_ok`, которая вернет признак успешного прохождения теста, если хотя бы две позиции в массивах будут отличаться. Наверное, для обеспечения хорошей реализации перестановки такой подход к тестированию может оказаться не слишком удачным, но это уже вопрос других тестов, разработку которых вы можете продолжить самостоятельно.

```
BEGIN{ use_ok( 'My::List::Util' ) }

use Test::More 'no_plan';

# # # # # # sum
ok( defined &sum, 'Подпрограмма sum() определена');
is( sum( 2, 2 ), 4, '2 + 2 = 4' );
is( sum( 2, 2, 3 ), 7, '2 + 2 + 3 = 7' );
is( sum( ), 0, 'вызов без аргументов дает 0' );
is( sum( -1 ), -1, '-1 = -1' );
is( sum( -1, 1 ), 0, '-1 + 1 = 0' );
is( sum( 'Ginger', 5 ),
    5, 'Строка + 5 = 5' );
is( sum( qw(Ginger Mary-Ann) ),
    0, 'Сумма двух строк дает значение 0' );

# # # # # # shuffle
ok( defined &shuffle, "Подпрограмма shuffle() определена");
my $array = [qw( a b c d e f )];

my $shuffled = $array;
shuffle( $shuffled );

my $same_count = 0;

foreach my $index ( 0 .. $#$array ) {
    $same_count++ if $shuffle->[$index] eq $array->[$index];
    }
cmp_ok( $same_count, '<', $#$array - 2,
    'Различия обнаружены, по крайней мере, в двух позициях');
```

Теперь, когда имеется набор тестов, можно приступить к написанию кода. В процессе написания исходных текстов модуля мы будем запускать тесты. Сначала большинство тестов будет терпеть неудачу, но по мере добавления нового программного кода все большее число тестов будет проходить благополучно. Ниже приводится конечный вариант модуля:

```
package My::List::Util;
use strict;

use base qw(Exporter);
use vars qw(@EXPORT $VERSION);

use Exporter;

$VERSION = '0.10';
@EXPORT = qw(sum shuffle);

sub shuffle { # алгоритм перестановки Fisher-Yates из perlfaq4
    my $deck = shift; # $deck - ссылка на массив
    my $i = @$deck;
    while ($i--) {
        my $j = int rand ($i+1);
        @$deck[$i,$j] = @$deck[$j,$i];
       }
```

```
    }

sub sum {
    my @array = @_;

    my $sum = 0;
    foreach my $element ( @array ) {
        $sum += $element;
        }
    $sum;
    }
1;
```

Ответы к главе 18

Упражнение 1

Поскольку речь идет о дистрибутиве модуля, который был создан в упражнении к предыдущей главе, у нас осталось не так много программного кода, который необходимо написать. Чтобы добавить возможность тестирования документации в формате POD, создайте файл `t/pod.t` (совершенно неважно, как вы его назовете) и добавьте в него следующие строки:

```
use Test::More;
eval "use Test::Pod 1.00";
plan skip_all => "Требуется наличие модуля Test::Pod 1.00 POD" if $@;
all_pod_files_ok( );
```

Назначение этих строк достаточно понятно: тест документации будет запущен только в том случае, если у пользователя установлен модуль `Test::Pod`.

Аналогичным образом можно добавить проверку с помощью модуля `Test::Pod::Coverage`, если на то будут серьезные причины. Создайте файл `t/pod_coverage` и вставьте в него программный код прямо из документации к модулю.

В зависимости от того, какие инструментальные средства применялись вами для создания модуля, все эти файлы уже могут иметься в наличии.

Упражнение 2

Для нового модуля с тестами можно было бы создать отдельный дистрибутив, но это совершенно необязательно. Вы можете просто включить его в состав дистрибутива, который уже был создан. Для этого достаточно сохранить файл модуля в нужном месте и добавить его в файл *Makefile.PL* или *Build.PL*.

Здесь мы не будем углубляться в объяснения и покажем лишь сам программный код. Проверку `$n == $m` выполнить не так просто, как может показаться на первый взгляд, но мы постарались упростить реализа-

цию настолько, насколько это возможно. Большая часть программного кода была взята прямо из примеров, приводившихся в главе, а затем была добавлена реализация функции sum_ok.

```
package Test::My::List::Util;
use strict;

use base qw(Exporter);
use vars qw(@EXPORT $VERSION);

use Exporter;
use Test::Builder;

my $Test = Test::Builder->new( );

$VERSION = '0.10';
@EXPORT = qw(sum_ok);

sub sum_ok {
    my( $actual, $expected ) = @_;

    if( $actual = = $expected ) {
        $Test->ok( 1 )
        }
    else {
           $Test->diag(
              "Ошибка в сумме\n",
              "\tПолучено: $actual\n",
              "\tОжидалось: $expected\n"
              );
              $Test->ok( 0 )
              }
          }

1;
```

Ответы к главе 19

Упражнение

Ну как, получилось? Мы и не ждали, что получится, поскольку еще в 1936 году Алан Тьюринг доказал невозможность общего решения. Более подробно о проблеме зависания можно прочитать в Википедии: *http://en.wikipedia.org/wiki/Halting_problem* (на русском языке: *http://ru.wikipedia.org/wiki/Halting_problem*).

В отличие от упражнений к предыдущим главам, здесь нам показать нечего. Теперь вы знаете, что такое тестирование, и знаете, что все делаете правильно, пока тесты проходят без ошибок (в противном случае тесты заканчивались бы неудачей).

Мы отпускаем вас в реальный мир. Нам нечего больше добавить к тому, о чем говорилось в этой книге. Теперь возвращайтесь назад и читайте сноски. Удачи!

Алфавитный указатель

Специальные символы

A

B

C

D

E

F

G

H

I

L

M

O

P

В

Г

Д

З

И

К

Л

П

Р

С

Т

У

Ф

Х

Ч

Ш

Э